Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Petr Kmoch

Exteriéry a interiéry ve virtuálních městech

Kabinet software a výuky informatiky
Vedoucí diplomové práce: Doc. Ing. Jiří Žára, CSc.

Studijní program: Informatika, obor Softwarové systémy, plán Počítačová grafika

Chtěl bych poděkovat docentu Žárovi za jeho vedení a rady, kterými mě směroval během práce, a za zapůjčení digitálního fotoaparátu. Také bych chtěl vyjádřit poděkování doktoru Lambertovi z Univerzity Nového Jižního Walesu za jeho volně použitelný kód pro výpočet Delaunayovy triangulace.

I would hereby like to thank associate professor Žára for his supervision and counselling which guided me throughout my work, and for lending me a digital camera. I would also like to express my thanks to doctor Lambert from The University of New South Wales for his freely usable code for Delaunay triangulation.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 8. srpna 2006 Petr Kmoch

Contents

Název práce: Exteriéry a interiéry ve virtuálních městech
Autor: Petr Kmoch
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: Doc. Ing. Jiří Žára, CSc., Katedra počítačů FEL ČVUT
e-mail vedoucího: zara@fel.cvut.cz
Abstrakt: Virtuální Stará Praha je webová aplikace umožňující prohlížení centra Prahy vymodelovaného ve VRML. Tato práce ji rozšiřuje o možnost prezentace interiérů a o impostory, což jsou zjednodušené části geometrie scény používané místo vzdálených modelů. Interiéry jsou založeny na existující struktuře VSP. Je přidáno automatické generování interiérových stěn, podpora pro stropy a osvětlení. Interiéry je možné doplnit do existujícího modelu bez nutnosti upravovat modely domů. Zároveň jsou do systému zavedeny hloubkové impostory. Mají plně trojrozměrný tvar vycházející z geometrie části scény, kterou zastupují. To umožňuje věrnější zobrazení rozdílů v hloubce v porovnání s klasickými plochými impostory. Každý impostor kvůli omezení velikosti používá jedinou texturu, předem vykreslený pohled na část scény, kterou zastupuje. Pro automatické generování impostorů je dodán program v jazyce Java. Pro zobrazování impostorů jsou zavedeny dva režimy, které se liší vizuální kvalitou a výpočetními nároky. Na závěr jsou uvedeny výsledky testů výkonu porovnávajících zobrazování impostorů s původním systémem.
Klíčová slova: virtuální realita, VRML, impostory, městské scény

Title: Exteriors and interiors in virtual cities
Author: Petr Kmoch
Department: Department of Software and Computer Science Education
Supervisor: Doc. Ing. Jiří Žára, CSc., Dept. of Computer Science and Engineering FEE CTU
Supervisor's e-mail address: zara@fel.cvut.cz
Abstract: Virtual Old Prague is a web-based application for browsing Prague's centre modelled in VRML. This thesis extends the project with support for interior areas and impostors, simplified pieces of geometry used in place of distant models. Interiors are based on existing VOP structure. Automatic generation of interior walls is added, as well as ceilings and lighting. Interiors can be added to the existing model without the need to remodel existing houses. Depth-augmented impostors are introduced into the system. Their geometry is a fully 3D shape based on actual geometry they replace. This allows for better displaying of depth discontinuities compared to a traditional flat impostor. A single texture, a pre-rendered image of replaced geometry, is used for the impostor, thus keeping its size small. A Java program is provided for automatic generation of impostors from the model. Two regimes are introduced for using impostors during browsing, with different visual quality and performance requirements. Results of performance tests are presented, comparing use of impostors to the original system.
Keywords: virtual reality, VRML, impostors, urban scenes

1 Introduction

1.1 Specification

This thesis presents an extension of the project Virtual Old Prague (VOP), which is a virtual reality presentation of Prague's city centre. The task of the thesis is to add two new pieces of functionality: presentation of interiors and using impostors for looking into the distance.

1.2 Virtual reality

Virtual reality is a term used for presenting data by trying to simulate reality in an electronic environment. This can range from rendering simple 3D scenes on computer screen to immersive applications fooling human senses with specialized hardware. Virtual reality can be used for simulating real-life situations (a virtual tour through a city or museum), interactive presentations (presenting an engine model or perhaps the structure of a molecule) or visualisation of abstract data like computation results. It has applications in research, education, military, industry and entertainment.

The key points of virtual reality are three-dimensionality, freedom of movement and interactivity. The ultimate goal of virtual reality is presenting a system indistinguishable from the real world. While this still remains the domain of fiction writers, extensive research continues in the field. From the point of view of approaching this goal, virtual reality systems can be classified into the following kinds:

Immersive VR This refers to the most perfect VR systems which make use of special hardware like head-mounted displays and 3D sound systems for presentation. User input is carried out via special outfits (gloves or whole suits equipped with sensors) or via external sensors (position tracking).

Augmented VR Augmented virtual reality combines real world with additional information. A typical example is a pilot's head-up display presented on their visor.

Projected VR QuickTime VR[1] is a typical example of projected VR. The basic idea is projecting panoramic images, allowing for a high-quality, all-around view. While rendering speed is high, the scenes are not actually 3D and thus user movement is limited to a pre-established set of points. Due to size of the panoramic images, files used can grow very large.

Multi-user distributed VR This approach to virtual reality focuses on interaction between users. It uses the concept of avatars, 3D objects (usually humanoid models) representing the user in the virtual world. A client/server architecture is used to allow the users currently present to see each other.

Low-end VR This refers to virtual reality systems which do not require any special hardware and can be rendered using a standard desktop computer. VRML and its successor X3D are typical examples of this approach.

Virtual Old Prague, the base of this thesis, is implemented in VRML. VRML stands for Virtual Reality Modeling Language. It is a language for specifying 3D scenes (called "worlds") in text format [2]. Worlds are then displayed using VRML browsers, which render the 3D world and allow

[1] A commercial system by Apple Corporation

the user to move about and interact with the scene. VRML is designed for easy usage on the Internet, so most VRML browsers exist as plugins into web browsers.

Since its creation, VRML has evolved. A new standard, X3D, has been introduced. X3D is based upon XML and offers a better mechanism for modularization and handling extensions. The traditional text format of VRML has been retained as one possible form of X3D syntax.

Even though the new standard is preferable, VRML is used in this thesis. This is due to the fact that the VOP project is written in VRML, as it dates to before X3D was introduced.

1.3 Virtual Old Prague

Virtual Old Prague was created as a student project on the Faculty of Mathematics and Physics of Charles University in Prague, under supervision of Doc. Ing. Jiří Žára, CSc. It is now maintained by the Czech Technical University in Prague, Faculty of Electrical Engineering. Present version can be viewed at `http://www.cgg.cvut.cz/vsp/`.

The original VOP allows the user just to move around the streets. It can be configured with parameters like detail level and visibility distance to tailor model complexity to the user's hardware capabilities. Offering reasonable browsing speed at a wide range of client machines was the primary goal of the project.

1.3.1 Structure & philosophy

VOP is built as a web-based application. While it contains a huge and complex model of Prague's centre, it is not too demanding on rendering resources, as the model is not downloaded in its entirety during browsing. The street network is divided into small areas called "sectors". Abstract visibility cost is defined between sectors and only those sectors visible from the one the user is currently in are downloaded and displayed. The visibility cost threshold is a parameter that can be specified by the user as part of customizing their browsing session.

A sector is polygonal in shape, its sides are referred to as "borders". A border can either contain one or more houses or it can be a so-called "gate" connecting two adjacent sectors. A gate is a complex mechanism which, when the user walks through it, initiates the process of discarding sectors no longer visible and downloading new ones.

The city scene is divided into separate models for each house or important object, which are stored in the server file system. The street plan, referred to as city topology, is stored in a database with references to these models, along with supplemental data.

Each house model can be supplied in multiple levels of detail (LODs). The system defines four levels, with level 1 being full geometry and increasing levels being of worse visual quality, up to a flat single-colour face at level 4. As part of session customization, the user can choose which level of detail the entire scene should be rendered in. Alternatively, geometry nearest to the user can be rendered at level 1, with higher levels being used with increasing distance of the model from the user, where losing details matters less.

The system is designed for viewing from ground level only. The vast majority of house models only contain their front sides and perhaps a part of roof visible from the street. This helps limit the amount of data required for download.

1.3.2 Important implementation details

Some implementation concepts of VOP are detailed here which are relied upon in the rest of this thesis. For a complete documentation of VOP, refer to [1]. Getting familiar with the implementation innards of VOP is advised for understanding the implementation of the program written as part of this thesis as well as changes made to the VOP itself.

Server structure The VOP server is a set of PHP scripts deployed on a web server. Models of houses and objects present in the city are stored as files in the server filesystem, but VRML files for the scene itself (i. e. combining the models into a world) are generated on the fly by these PHP scripts. These are the most important ones:

`sector.php` This script produces the VRML file for one sector. Models of houses in the sector are included in it via `Inline` nodes. The entire file is encased in a `Group` node, whose first child is a parameter node (instance of custom prototype `PAR`) which carries the sector's identifier. The world-controlling script (see below) relies on the fact that the sector file is enclosed in a single node and that the parameter node is its first child.

`surround.php` This script retrieves data related to sector switching from the database. Each pair source sector–destination sector has its own parameters, which include sectors to be loaded and discarded and gate positions in the destination sector. This data is encapsulated in VRML to be usable by the world-controlling script.

`vrml.php` The VRML file generated by this script represents the entire scene. It contains a root node for all geometry and all sectors are added to the world as child nodes of this root. The most important part of the generated file is a `Script` node called `SCR`. This world-controlling script implements all the logic behind sector switching and management. It is responsible for downloading new sectors and discarding those no longer needed. It also operates the head-up display.

Sector-switching mechanism The user can only walk from one sector to another through a gate. Each gate is an instance of a prototype called `PROXG`. An important aspect of gates is that they are not part of any sector. Instead, eight global instances of gates exist, and these are positioned on the relevant borders of the active sector when the sector is entered. When the user walks through a gate, the gate sends an event to `SCR`.

`SCR` knows which sectors to load and which to discard from the sector-switching parameters obtained from `surround.php`. Sector loading is initiated via function `load_sect`. Upon receiving sector data, it calls `done_sect`, which inserts the sector into the scene graph. Sectors are removed from the scene via function `del_sect`. Rather than discard the sector altogether, it is inserted into a sector cache. The cache is organized as LRU and its implementation is quite complicated. `load_sect` first searches for the sector in the cache and only when it is not found there it asks the server (`sector.php`) for its data.

Database City topology and geometry information is stored in the tables `sector`, `border`, `vertex`, `lnode` and `node`. Table `sector` contains the sector's ID and information about its surface.

A sector is a polygon whose points are stored in table `vertex`. Borders are specified in table `border`. It contains a reference to the sector to which the border belongs and the two vertices which constitute the border. These are referred to as a and b. Note that borders are orientation-sensitive — AB and BA represent two distinct borders. This notion is very important for the impostor placement process introduced in this thesis. Borders on a sector's edge must appear in counter-clockwise order (seen from above). That is, only a border's left-hand side is normally designed for being viewed.

A border can be of type '`wall`' or '`proxi`'[2]. A '`proxi`' border represents a gate, as mentioned above. A border of type '`wall`' denotes geometry — it can contain one or more "`lnodes`" (linked nodes). Each `lnode` represents a building and has its own geometry file.

A `node`, often referred to as "free-standing node", is a piece of geometry not linked to a border. Free-standing nodes include objects like trees, lamps, tram stops and fountains. Each node has its own VRML file.

1.4 Goals of this thesis

Two tasks were set in the specification of this thesis — adding interior presentation and impostors to the VOP system. Both of these tasks were accomplished. Interiors are dealt with in chapter 2, implementation of impostors is detailed in chapters 3 and 4.

Throughout the implementation, it was attempted to conform to the existing structure and conventions of VOP, to maintain consistency in code in terms of identifier selection etc. Care was taken to ensure the new system can be configured to behave exactly like the old one (i. e. to make it possible to disable all functionality added). Changes to existing database structure were minimized in favour of adding new tables, to prevent accidentally breaking the functionality of existing VOP code.

VOP also comes with several tools for designers of models for the virtual city. These were not updated to reflect the additions made and likewise, no new user-oriented authoring tools were created. For impostors, a simple program which offers design functionality was created (see chapter 4). Interiors presently have to be entered into the database by hand. Creating complex authoring tools in addition to implementing the presentation systems themselves would probably require a team of programmers and was thus considered beyond the scope of this thesis.

[2]Other border types exist, but these are not relevant now

2 Interiors

One part of the work to be accomplished by this thesis was to allow the user navigating the virtual world to enter houses, courtyards and similar interiors. Two main steps were needed to facilitate this: a mechanism to perform the transition between the exterior and an interior area, and means to present the interior visually.

2.1 Interior properties

Since courtyards are in a way already present in the original VOP, the main focus was put on building interiors. Common building interiors seem to share the following characteristics:

- They are partitioned into small blocks (rooms, corridors) with few and precisely-defined points where moving from one block to another is possible. Also, usually only the immediately adjacent blocks are visible from a given block.
- The blocks' boundaries are formed by vertical walls, either of flat colour or bearing repetitive designs, such as wallpapers, ornaments or masonry.
- Blocks have ceilings, which share much of the characteristics of walls, though they sometimes have more complex geometry.
- Actual objects found in interiors vary enormously, though some architectural features like windows, stairways and doors are common.
- Windows and doors present a link between an interior and its surrounding exterior area.

Based on these observations, it became apparent that interiors share many of the properties of exterior areas as they are handled by the VOP. The one thing most obviously missing is support for ceilings.

2.2 Introducing interiors into the VOP

2.2.1 Blocks

When designing an interior, a block such as a room or corridor is represented as a `sector`. This allows for VOP's visibility mechanism to be used. On the implementation side, it brings a major saving, as the procedures already present for loading, displaying and caching sectors can be used without modification to manage interiors as well.

2.2.2 Walls

The system of `lnodes` as present in the VOP could also be used to represent interior walls[3], but it is obviously ill-suited for this. A typical `lnode` is a flat face representing the front side of a house (possibly with indented windows) with an arbitrarily-shaped top. Depending on the LOD, it is either textured with a single large texture (a photo of the house), or contains many additional small faces, each displaying a texture (usually of a window).

Interior walls have different requirements. As has been stated, they are usually flat-coloured or could be easily textured by repeating a small texture. More importantly, their shape is very much

[3] Interior borders would then be of the existing type '`wall`'

enforced by the floor and ceiling, and it would thus be pointless to require modellers to model their exact contours. Instead of using `lnodes`, a new type of border, '`inwall`', was introduced.

The geometry for an `inwall` border is generated dynamically when the sector is loaded (for details, see section 2.3.2). Each `inwall` can be assigned a colour and/or a '`wallpaper`'. A `wallpaper` is an entity similar to a texture prototype. It represents a texture and two scaling parameters, `size_s` and `size_t`. As default, the texture is repeated on the generated wall and these parameters determine the size (in metres) of the texture's image. It can also be specified that the texture should not be repeated in one or both axes. In such a case, the texture is scaled to fit the generated wall exactly.

`inwalls` are generated in such a way as to be compatible with the LOD system present in VOP. An `inwall` with a `wallpaper` is treated as detail level 2, an `inwall` with colour as level 4. If both are specified, the system can generate both versions and use them as LODs as appropriate[4].

While most interior walls conform quite well to the above scheme, some don't. Forcing all interior walls into being flat and bearing either a texture or colour was considered as too restrictive. Therefore, the system also allows VRML files to be specified for `inwalls`. These follow the standard naming convention used in VOP. In practice, they are most useful for providing detail level 1 or 3 for an interior wall. However, if a file for level 2 or 4 is specified, it is used instead of the generated geometry (i. e. files take precedence). For example, an ornament-decorated wall with a small alcove could be created by specifying a `wallpaper` for the ornaments, a colour and a level 1 file containing the entire wall with the alcove. The wall would then have three valid LOD levels; level 1 from the file and levels 2 and 4 generated by the system.

It is assumed that interior walls will be created by using `inwall` borders. However, nothing stops the designer from assigning any other border type (like `wall`) to a border representing an interior wall. Not imposing any limitations on this was arbitrary, to give designers more freedom when creating exotic interiors. It is also possible to go the other way round and use an `inwall` border for a non-interior setting. However, care has to be taken, as the `inwall` generation mechanism relies on ceiling data being specified. See section 2.2.4 for details.

2.2.3 Windows and doors

From a wall's point of view, windows and doors are very similar. They are openings, usually rectangular in shape, which allow the user to see beyond the wall. For a border of type `inwall`, it is possible to specify one or more windows or doors (collectively referred to as '`holes`' in the `inwall` generation algorithm) contained in that border. The position of a `hole` is specified as an offset from the border's starting point. Width and height are also specified, as well as elevation of the `hole`'s bottom edge from the border's ground level. The elevation is primarily useful for windows, but it can also be used for doors, for example when a door is located above a short flight of stairs.

Doors serve an additional function compared to windows — they allow the user to walk through them and enter different parts of the interior. This translates as entering a different sector. The existing system of gates was used to facilitate this. A gate (`PROXG`) is placed on the position of each door specified[5]. This approach allows the entire sector-switching mechanism to be used on interiors without modification. The destination sector is specified as a parameter of the door.

[4] For details on the LOD system, see [1].

[5] Note that this limits the number of doors in one sector to the number of gates supported by the VOP system (currently 8).

For the sake of the wall generation algorithm, only rectangular windows and doors are allowed. This is no serious limitation, as the vast majority of real-world windows and doors conform to it. It is possible to design walls with arbitrarily shaped openings by hand, bypassing the generation mechanism. It is also possible to create a wall with fancy-shaped windows as LOD level 1, and have rectangular windows in worse quality levels generated automatically.

The same holds true for doors. However, in order to have the sector-switching mechanism work properly, the door must be specified in the system (in the database) even if it is not used in the wall generation process (i. e. the wall is not generated automatically).

Windows and doors also allow the user to see through the wall they are in. This breaks into two major cases: seeing into a different part of the interior and seeing out of the interior altogether.

Looking into the same interior Openings which reveal another part of the same interior are usually doors, as windows between two adjacent interior blocks are rare. This simplifies the matter, as it means the sector visible through the door must be already loaded. There is thus no problem in letting it be seen.

It was considered whether impostors could be used in doorways, as they are on exterior gates. However, the overall scale of interior areas is much smaller than that of exterior ones. Impostors for such close-up use have to be of better quality than those used in exteriors, as was also shown in [6]. Adding such an extra level of complexity would complicate and slow down the already time-expensive impostor generation algorithm (see section 4.4.3). Also, the algorithm would have to be modified to include free-standing `nodes` into the impostor geometry, as they form a substantial part of an interior's appearance. Rather than complicate matters that much, it was decided not to use impostors in interiors.

Looking outside In contrast with seeing into the same interior, the surroundings of a building are primarily visible through windows. They are thus inaccessible and therefore it is unnecessary to keep their complete sector information in memory. Here, impostors were found to be the ideal solution. They represent much simpler data than complete sector geometry and offer a view without distance limitations, which is just what the user expects. Surroundings seen through windows are mainly perceived as background, just 'at a glance'. Under these conditions, limited visibility distance would hamper visual quality more than low detail level.

2.2.4 Ceilings

Ceilings are a major feature of interiors, and one not present in the original VOP at all. However, in many ways, a ceiling is identical to a floor. With this in mind, it was decided to implement ceilings by adapting the current mechanism for rendering floor. This mechanism constructs the floor as a surface from a series of points with optional triangulation and appearance information. These points include all the vertices of the sector edge, plus any additional points specified as part of floor definition by the designer. This mechanism clearly suits most ceilings as well. However, one important piece of information missing is wall height.

In order to implement ceilings, several attributes were added to the database. It is possible to specify whether a `sector` has ceiling, and to specify the ceiling in exactly the same format as that used for floors. Each `vertex` can also be assigned a wall height, representing the height of the ceiling above this vertex. The scripts assume that if a sector is specified to have ceiling, all its vertices have wall heights specified. This wall height information is then used to compute the points forming the basis of the ceiling (analogous to the vertices on sector edges for floor) as well as during `inwall` generation.

The surface rendering mechanism used can theoretically create arbitrarily complex ceilings, but it does have downsides. All the geometrical data must be stored in the database, and, most importantly, all textures to be used on the ceiling must be prototyped and defined as `appearances`. This is OK for mostly flat ceilings of constant colour or bearing a repetitive texture. For really complex ceilings, like a cathedral's, for example, this approach becomes too cumbersome. In such cases, it would be more practical to specify the ceiling using a VRML file.

The existing mechanism of free-standing `nodes` was used for this. To create a sector with a complex ceiling, the designer marks it in the database as not having a ceiling[6]. Then the ceiling is created and inserted into the model as any other `node` would be. This means that such ceilings are subject to the LOD mechanism as normal. Note that even for such a sector, it is possible (and probably desirable) to specify wall heights in vertices. These will be used for generating the shape of `inwalls`.

2.2.5 Entry/exit points

In the real world, an interior is accessed by passing through a door, similar to one found in interiors. However, in the VOP, it would be impractical to use the mechanism for interior doors (described in section 2.2.3) to implement interior access doors as well. There are two main reasons for this.

First, it is desirable to clearly separate interiors from exteriors. Thus, when inside an interior, there should be no exterior geometry loaded. The door mechanism uses gates, and would thus require the first interior sector entered to be adjacent to an exterior sector, so both would have to be in memory when passing through the door.

Second, using the door mechanism would require re-designing the `lnode` of a house when an interior was created for the house, which is obviously unnecessary.

There would be other drawbacks to this approach, too. The system of doors would have to be expanded to work on borders other than `inwalls`, and the door openings themselves would cause unnatural-looking holes in the house when viewed from such a distance that the interior sector would not be loaded.

Clearly, a separate mechanism for accessing interiors was needed. A VRML prototype named "`Doorway`" was designed for this. It is an invisible, clickable piece of geometry coupled with a proximity sensor. When the user is close enough to the doorway, they can click it and the system will transport them into the associated interior (or vice versa). A doorway is rectangular in shape, with its size freely specifiable in the database. Each doorway is linked to a `border` or an `lnode`, with its position specified as an offset from its start. Elevation of the doorway's bottom edge from the border's level can also be specified. The intended use of a doorway is to place and size it so that its clickable area lies over the image of the real-world door in the house's model[7].

A doorway's destination is referred to as an `entrance`. An `entrance` consists simply of a point in 3D space, camera orientation and sector number. It is basically identical to a VOP viewpoint (a `location`), but it was decided to use a separate entity for `entrances`. It would seldom make sense for the user to select an `entrance` as their starting point of browsing the city model, and the number of `entrances`, two for each real-world doorway[8], would clutter the list of viewpoints.

[6] Which actually means not having an automatically generated ceiling.

[7] Or in the model of an interior wall. `Dooways` are used for exiting interiors as well as entering them.

[8] One each for the interior and exterior side

2.2.6 Furniture and other objects

The VOP already contains the mechanism of free-standing `nodes`, which can be used to add an arbitrary object into the model. Due to the variability of potential objects found in interiors (as pointed out in section 2.1), no specialized system for them could bring substantial benefits over the generic one. Thus, it was decided to leave the entire support for furniture and architectural phenomena like stairs or pillars on the `node` mechanism.

2.2.7 Lighting

The original VOP contains no lighting at all, as it doesn't require it. House exteriors are represented by textures and thus aren't subject to lighting. Where textures are not available, colour faces are defined with `emissiveColor`, and thus require no lighting either.

This is unsuitable for interiors, where large surfaces of flat colour are common. The absence of lighting makes it impossible to discern face boundaries, e. g. tell the wall from the ceiling, when they are of the same colour. Point lights and spotlights can be inserted into the scene via the `node` mechanism. However, this does not hold for directional lights, which are probably best suited for lighting interiors, as they can easily simulate sunlight coming in through windows. The `node` system inserts each object into its own `Transform` VRML node, which renders directional lights useless[9]. Thus, it was necessary to allow directional lights to be added into the system.

It was decided to link directional lights to sectors. This makes it easy to place the light's VRML node on the correct position in the scene graph. Also, each sector can thus have its own light parameters, which is useful. The system allows for all of the light's attributes to be set, namely colour, direction, intensity and ambient intensity. Each sector can only be assigned one light, and of course can be left without a light altogether. Directional lighting is not limited to interior sectors, though there it is most useful.

2.3 Interior implementation

This section describes the implementation details of the interior-related features described in section 2.2.

2.3.1 Changes to the database

Support for interiors is the only part where changing the structure of existing database tables was necessary. In the table `border`, a new value, '`inwall`', was added to the field `type`. All the other changes were accomplished by adding new tables to the database. A list of these tables follows.

`entrance (id, sector_id, x, y, z, orient)`

> This table contains information about entrances. For each entrance, it specifies its position, view direction and the sector it is in. Entries from the `ldoorway` table refer to entrances by their `id` field. Entrances and doorways are discussed in section 2.2.5.

`indoor (id, border_id, offset, elev, width, height, neighbour)`

> Each entry in this table represents a door connecting two interior sectors. The door is present on the border `border_id`. It is a rectangle with the specified `width` and `height`. `offset`

[9] See [2], section 6.18

and `elev` specify the position of its lower right-hand corner (when looking at the border). `offset` is the distance from the border's starting vertex, measured in the XZ plane. This is similar to the `offset` field in table `lnode`. `elev` (and `height`) is only used for rendering the door, that is, when the border specified is of type `inwall`. It specifies the elevation of the door's lower edge above the border's starting vertex, measured along the Y axis.

A gate (`PROXG`) is placed on the position of each door, matching its width. `neighbour` specifies the destination sector. See section 2.3.2 on how door data is used for wall generation.

`inwall (id, color, wallpaper_id, koef, filename)`

This table holds additional information for borders of type `inwall`. `id` is a reference into the table `border`. Both the fields `color` and `wallpaper_id` are optional, though at least one of them should be specified. `color` contains the wall's colour, used for generating LOD level 4. It is interpreted as a 3-byte number specifying the red, green and blue components, with blue being the least significant byte. `wallpaper_id` is a reference into table `wallpaper` and is used to generate LOD level 2. `koef` has the same meaning as in table `lnode`, namely the scaling factor of distance at which different LODs are swapped. `filename` specifies the directory containing geometry files for one or more LODs of this wall. It can be left blank. Data from this table is used to generate interior walls, as outlined in section 2.2.2.

`ldoorway (id, line_type, line_id, offset, width, height, elev, entrance_id)`

This table specifies doorways, which are used to transfer the user between interiors and exteriors. Each doorway is linked with a '`line`', either an `lnode` or a `border`, as specified by the `line_type` field. `line_id` then references the appropriate table. `offset`, `width`, `height` and `elev` have identical meaning as those in table `indoor`. Only for doorways linked to an `lnode`, the position of the lower right-hand corner is relative to the `lnode`'s starting point, not the border's. `entrance_id` specifies the destination entrance of this doorway as a reference into table `entrance`. Use of doorways and entrances is described in section 2.2.5.

`sector_flags (sector_id, flags)`

This table is designed with future expansions in mind. It allows different flags to be specified for sectors, which are indetified by `sector_id`, a reference into table `sector`. Currently, it contains only one possible flag, '`interior`'. Sectors marked with this flag are considered interior when such a distinction is called for. Currently, this only applies in the impostor placement algorithm (see section 4.4.4).

`sector_interior (id, has_ceil, ceil)`

This table specifies ceiling data for sectors. `id` is a reference into table `sector`. `has_ceil` and `ceil` are analogous to fields `has_surf` and `surf` in that table, but refer to the ceiling rather than floor. Ceiling generation is described in section 2.2.4.

`sector_light (sector_id, color, intensity, ambient, x, y, z)`

This table specifies directional light present in a sector. `sector_id` identifies the sector in table `sector`. `color` specifies the light's colour. It is interpreted as a 3-byte number specifying the red, green and blue components, with blue being the least significant byte. `intensity` and `ambient` specify the values for the directional light's `intensity` and `ambientIntensity`

VRML fields, respectively. `x`, `y` and `z` are the components of the light's direction vector. Sector lighting is discussed in section 2.2.7.

`surround_impostor (sector_id, impostor_id)`

This table specifies which surround-type impostors are to be used in which sector. More than one impostor can be specified for a sector, and an impostor can be used in more than one sector. `sector_id` references table `sector` and `impostor_id` table `gener_impostor`. Use of surround-type impostors is discussed in section 2.2.3.

`vertex_height (id, height)`

This table contains additional data for some vertices. `id` identifies the vertex in table `vertex`. `height` gives the height of the ceiling in this vertex. Heights are used for ceiling and interior wall generation; refer to sections 2.2.4 and 2.2.2 for details.

`wallpaper (id, texture_name, size_s, size_t)`

Each entry specifies a wallpaper. `texture_name` is the name of the wallpaper's file, which must be present in folder `vsp/wallpapers`. `size_s` and `size_t` determine the scale of the texture along the S and T axis, respectively. A size of 1.0 means the texture image is considered to be of size 1 metre. Specifying a size of 0.0 for an axis prevents the texture from being repeated along that axis. Instead, it will be stretched or shrunk to fit the wall exactly. Wallpapers are used for interior wall generation, as outlined in section 2.2.2.

2.3.2 Changes to script `sector.php`

Several changes were necessary in the sector-generation script. Support for generating ceilings and interior walls was added. Placement of doorways is also governed by this script, as well as sector lighting.

Ceiling generation The process already present for generating sector floor was re-used with minimal modifications to generate the ceiling as well. Some changes had to be made to the classes in `surface.php` to accommodate for the fact that more than one default appearance can exist, namely one for the floor and one for the ceiling. The default appearance for the ceiling is not a texture, but a flat light grey colour.

Interior wall generation For a border of type `inwall`, the system can use files or generate geometry as necessary. Files for any LOD can be present. Generated geometry with a wallpaper is treated as LOD 2, geometry with flat colour as LOD 4. `inwalls` are subject to the LOD mechanism just like `lnodes`, so one or more LODs can be inserted into the scene as required. If LOD of level 2 or 4 is required for the scene and no file is specified for it, the system will generate it if a wallpaper or colour is specified, respectively.

If both levels 2 and 4 are being generated, they share their geometry data via a `DEF/USE` construct. The geometry is generated by algorithm 1.

The goal is to divide the wall up into quads so that all windows and doors, collectively referred to as "holes", are at their right positions. First, all holes in the border are retrieved from the database and sorted according to their position in right-to-left, bottom-to-top order. The wall is then divided into vertical stripes, a dividing line placed on both vertical edges of each hole. Each stripe is then processed separately.

Algorithm 1 Interior wall generation

```
holes=getHolesFromDatabase(borderId);
sort(holes,"offset, elev");
stripes=splitByVerticalEdges(holes);
for each stripe in stripes {
  strata=splitByHorizontalEdges(stripe);
  for each stratum in strata {
    if nonHole(stratum) geomery.addQuad(stratum);
  }
}
return geometry;
```

The stripe is divided into horizontal “strata”. Dividing lines between strata are placed at horizontal edges of holes present in the stripe. Unless holes intersect, this yields alternating ‘wall’ and ‘hole’ strata. ‘Wall’ strata are then added to the generated geometry as quads.

This algorithm ensures that the wall is properly divided into quads, which leads to a simple triangulation. The wall is then rendered as an `IndexedFaceSet`. An example of dividing a wall into stripes and strata is given in figure 1.

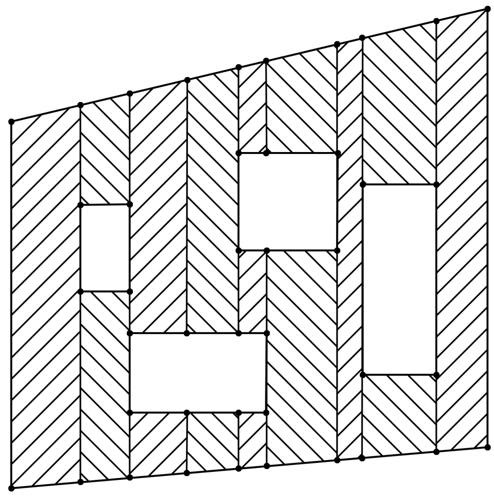


Figure 1: Quads generated for an interior wall

For common walls, algorithm 1 would actually produce less quads if the wall was divided first into horizontal strata and these into vertical stripes. However, this would complicate the algorithm considerably. While a wall’s left and right edges are always vertical, its top and bottom edges can be angled. This happens when the sector’s floor or ceiling is sloped, as is common for stairways. Having the edge corresponding to the major axis angled could cause some quads to degenerate into triangles (see figure 2) and would require many extra checks in the algorithm. Overall, simplicity of the generation algorithm was preferred.

Sector lighting The addition of the per-sector directional light was pretty straightforward. If a light is specified for the sector, a `DirectionalLight` VRML node is inserted into the sector scene graph’s topmost `Group` node, the one containing the sector’s `PAR` node as well as all its geometry. This ensures that all objects present in the sector are affected by the light. All parameters of the

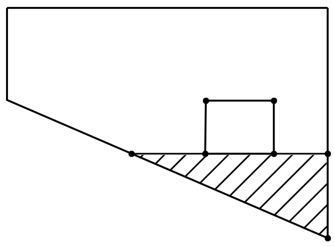


Figure 2: Quad degenerated into a triangle under horizontal-major approach

light are taken from the database.

Doorways For doorways, a new VRML prototype called "`Doorway`" has been added to the generated sector file. The prototype encapsulates a transparent, non-collidable box, a touch sensor, a proximity sensor and a short controlling script. When the user enters the proximity sensor's area, the touch sensor is enabled and the doorway becomes clickable. The proximity sensor is included so that the user doesn't enter a distant doorway by accidentally clicking on it. Also, being able to enter a door tens of metres distant could disturb the browsing experience.

Clicking the doorway generates an `eventOut MFFloat set_location` event on the `Doorway` instance. Several pieces of information have to be transmitted at the same time, so `MFFloat` was chosen as the event type. The content of the generated array is: `[ sector_id, x, y, z, orientation ]`. It represents the destination entrance. The data for it is retrieved from the database and stored in the `Doorway`'s fields when the sector is being loaded.

This event has to be routed to the entire world's main `SCR` script somehow. To facilitate this, an `exposedField MFFloat new_location`[10] was added to the sector's `PAR` prototype. Each doorway in the sector then routes its `set_location` event to this `new_location`. The main `SCR` script node can access the sector's `MY_PAR` node and read the `new_location` event. See also the changes in `vrml.php`.

2.3.3 Changes to script `vrml.php`

Two changes in `vrml.php` were necessary to support interiors, moving the user when they click a doorway and facilitating the display of surround-type impostors. All other interior-related features are either handled in `sector.php` or use the standard mechanisms present in VOP.

Doorways When the user clicks a doorway, the sector containing the doorway generates a `new_location` event on its `MY_PAR` node. To process this event, `eventIn MFFloat set_entered_location` was added to the script `SCR`. When a sector is loaded (in function `done_sect`), a route is dynamically established from its `new_location` to `SCR`'s `set_entered_location`.

At first, `set_entered_location` was implemented simply as a wrapper for calling the already present viewpoint-switching mechanism, as a viewpoint and an entrance have identical parameters. However, a problem arose, as the process of switching to a new viewpoint includes removing all currently loaded geometry from the scene. Most of the time this would be desirable, as interior

[10]The `exposedField` is used as a "pipe" to receive data from within the sector and pass it onward. The stored value is not used for anything.

and exterior sectors are not displayed at the same time. However, when the session is configured to show the whole model (`visibility` is set to '`all`'), entering an interior caused the entire world to reload. This was of course unacceptable and so `set_entered_location` was altered. It is still based on the viewpoint-switching mechanism, but it bypasses the geometry sweeping. Also, when the sector being entered is already loaded, it also bypasses the blackout/blackin effect. However, the core of the sector-switching mechanism, function `set_new_sector`, is still used in `set_entered_location`.

Surround-type impostors Each sector can define any number of surround-type impostors to be displayed together with the sector. Each surround-type impostor is placed on a border. Unlike proxi-impostors, surround-type impostors do not participate in the sector imposting mechanism (see section 3.1.2 for details), so their management is considerably more simple.

The mechanism for dealing with surround-type impostors is very similar to the one for dealing with sectors. Which surround-type impostors are to be added or removed is governed by the script `surround.php`. The actual addition and removal is performed by the `SCR` script node.

The management of loaded surround-type impostors is similar to that of loaded sectors[11] and is contained in the fields `a_surrimp_loaded`, `i_surrimp_loaded` and `surrimp_req`. When a surround-type impostor is added to the scene, it is also added to the end of the `a_surrimp_loaded` array. Its index is remembered in the associative array `i_surrimp_loaded`. Deletion is handled differently than for sectors. Rather than keep a list of holes in the `a_surrimp_loaded` array, the array is shortened by one upon removing an element. The element to be deleted is replaced by the last element in the array, whose entry in `i_surrimp_loaded` is updated to reflect its new position.

Loading of surround-type impostors is initiated in the function `done_par`, according to the sector-switching parameters, in a manner similar to loading sectors. Surround-type impostors are generated by a new script, `surround_impostor.php`, which is described in section 2.3.5.

There is also a cache for surround-type impostors, where they are placed when removed from the scene. To avoid the complexity of a true LRU cache as present for sectors, the cache for surround-type impostors is simply FIFO, implemented as a cyclic array. In order to allow this, impostors are not removed from the cache when added to the scene. At the core of the cache is an associative array formed by a `a_surrimp_cached`, `i_surrimp_cached` pair. An integer field, `surrimp_cache_start` is used to point at the oldest element in the cache (i. e. the one to be removed next). When a surround-type impostor *imp* is removed from the scene, it is added to the cache. If the cache is already full, the oldest element is replaced by *imp* and `surrimp_cache_start` is increased by one, looping back to 0 if the array's end is reached. The cache can hold 8 surround-type impostors. This number was chosen to allow for four views out of the building and perhaps four views into a courtyard or similar area, and should thus be sufficient for most interiors whilst still keeping the cache's memory footprint small. The cache is designed to speed up loading when walking around one interior; keeping surround-type impostors for separate interiors is not its purpose.

2.3.4 Changes to script `surround.php`

`surround.php` specifies the parameters of switching from one sector to another. Two changes were necessary, determining surround-type impostors to add/remove and allowing for gates to be placed on interior doors.

[11] See [1] for details of sector management.

Surround-type impostors Two `MFNode` fields were added to the `PAR` prototype, `addSurrImp` and `removeSurrImp`. These contain the numbers of impostors to be added to the scene or removed from it, respectively. They are computed as set differences between surround-type impostors visible from the sector being entered and the one being left.

Doors The standard mechanism of determining gate positions was used to place gates on doors. Relevant door data is retrieved from the database and gate positions are calculated accordingly. They are then added to the arrays already used for gates on `proxi` borders. This means that the sum of `proxi` borders and interior doors present in a sector is limited by the number of `PROXGs` available in the system, which is currently 8.

2.3.5 New script: `surround_impostor.php`

This script is responsible for generating a VRML file for a surround-type impostor. Its function is similar to that of `sector.php`, but it is considerably less complex.

The generated VRML file is fairly simple. It contains only a `PAR` node identifying the impostor and an inline of the impostor file as specified in the database. The structure of the generated file is as follows:

```
PROTO PAR [
 exposedField SFString my_id ""
] {
 Group {}
}
Transform {
 place_impostor_on_border
 children [
  PAR { my_id "impostor_database_id" }
  Inline { impostor_file }
 ]
}
```

Note that surround-type impostors are not subject to any further `Transforms`. They are inserted into the scene just as this script sends them. Therefore, this script retrieves the position of the impostor's border from the database tables `border` and `vertex` and translates & rotates the impostor accordingly.

3 Impostors

An impostor is a simplified representation of a part of the scene, used in place of the part's actual geometry when visual quality is not too important. Unlike a model with lower level of detail, an impostor usually replaces more than one object and its geometry need not correspond to the geometry of the replaced objects at all. The simplest and perhaps most typical impostor is a flat face or "billboard" with a texture which holds a pre-rendered image of the objects replaced. In common use, the word "impostor" usually refers to such a billboard.

As discussed in [5, 6], such impostors offer very poor visual quality. In practice, they can only be used for views from points very close to the point their texture was rendered from, otherwise their nature becomes immediately obvious and detracts from the overall visual quality of the scene. As an upside, though, they offer the simplest solution, both in terms of geometry complexity and texture size.

A better approach is suggested in [5]. An impostor's texture stays the same, a pre-rendered image of the part of the scene to be replaced. But the geometry is augmented by depth information obtained from the Z-buffer during the texture's rendering. Thus, the impostor is no longer a flat face, but a fully 3D mesh. This improves the overall visual quality of the impostor and allows for self-occlusion and parallax effects, making the impostor usable under a greater range of viewing angles and distances. Points on depth disparity lines are included in the mesh, amplified by a rectangular grid to achieve finer triangulation. Constrained Delaunay triangulation is used to construct the mesh. The example application given is an urban scene.

This method seemed intriguing and it was decided to try to implement a similar approach for the VOP. A Java program was created to perform impostor generation and other design-time tasks, which is described in chapter 4. The rest of this chapter is devoted to changes made to the VOP to enable impostor displaying and generation.

3.1 Displaying impostors

3.1.1 Position

The first consideration to be made was how to connect impostors into the current scene structure. Impostors are used in places where the user can see into the distance. Such places naturally correspond to gates between sectors. Thus, it was decided to place impostors on `borders` representing gates.

3.1.2 Impostor–sector interaction

Each impostor represents the geometry of one or more sectors. The impostor is said to "impost" those sectors. Clearly, it must be ensured that an impostor *imp* and a sector imposted by *imp* are never displayed at the same time. At the very least, it would be a waste of resources. More importantly, their geometries are present at the same position in the scene, so they would clip or permeate each other.

Two simple solutions would be to either always prefer the impostor or always prefer the actual sector when both are potentially displayable. Neither of these is optimal, however.

The first one, to always prefer the impostor, is obviously wrong. Impostors are view-dependent and even though the depth-augmented impostors used offer a greater range of viewing angles, they certainly cannot be viewed from the side, let alone from opposite the intended viewing direction.

Yet this is precisely what could happen if the impostor was always preferred, as illustrated in figure 3.

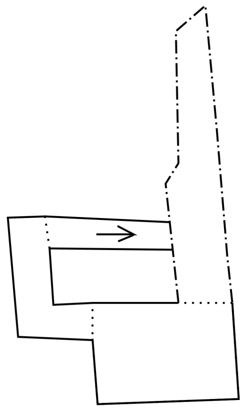


Figure 3: User reaching an impostor's side

The other approach, only displaying the impostor if the sector is not available, doesn't have this problem. However, it decreases the overall usefulness of impostors, as they are only displayed on the boundaries of the currently visible part of the scene, even though there might be impostors available for nearer parts of the scene.

The trouble is that in the general case, an optimal solution would require complex data structures and procedures for evaluating the user's current position in relation to all sectors loaded. It would be even more difficult to implement such an algorithm with the limited options available inside VRML `Script` nodes.

Impostors are good for replacing distant geometry, but their low visual quality becomes obvious when viewed close-up. This means that they have to be hidden and the imposted sectors restored when the user comes close enough. This would further complicate the general algorithm for determining impostor/sector preference. An example of a complicating situation is given in figure 4. The user has approached the hatched impostor, meaning it has to be hidden. The algorithm would have to find some means of discerning the difference between sectors A and C, as A has to be visible, while C can safely be imposted. Sector-to-sector visibility cannot be relied on, as these numbers are chosen arbitrarily and are not subject to any constraints.

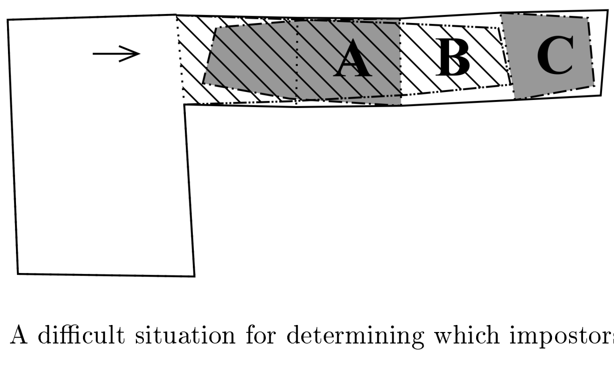


Figure 4: A difficult situation for determining which impostors to show

Rather than introduce a complicated general algorithm that could possibly slow browsing down, it was decided to impose restrictions on impostor displaying. Using impostors was split into two regimes, one that uses an optimal algorithm with extra constraints specified on visibility and another one which uses the "always prefer the sector" approach.

The first regime, called “limited visibility regime”, tries to use impostors to their full potential, using them instead of actual geometry whenever possible. However, to bypass the complexity of the general case as outlined above, sector-to-sector visibility is limited to immediately neighbouring sectors. The algorithm used for determining which impostors to show or hide is detailed in section 3.4.2. This regime is good when there are plentiful impostors available, that is, when almost all gates have an impostor. If this condition is met, the limited visibility matters little, as geometry beyond the immediately neighbouring sectors is included in the impostors displayed. Figure 5 demonstrates this regime. Imposted sectors are greyed, impostor area is hatched. Dashed sectors are not loaded. Scene (a) is a situation with plentiful impostors, where visual quality is not hampered. This regime is less suited for the scene (b), where a lot of the gates have no impostors and thus the limited visibility shows.

The other impostor usage regime allows for any visibility setting. All sectors visible are always rendered in full geometry and impostors are only displayed for sectors beyond the visibility range, that is, only on the edge of the loaded part of the world. While this regime can be configured to offer good visual quality even in parts where impostors are not present, it is more expensive in terms of rendering resources. This regime is demonstrated in figure 6. The same scenes are shown as in figure 5, to allow for easy comparison of the two regimes.

3.2 Impostor generation

Impostors are pre-rendered, that is, they are generated at world-design time and stored in the database and server filesystem. For reasons discussed in section 4.1, impostors are generated so that rendered images are obtained by capturing the screen output of a VRML browser. These are combined with topology data retrieved from the database.

An impostor is generated as a view from a fixed point in the scene, the “generation point”. This point is determined by a separate process called “impostor placement”, which is described in section 4.4.4. Impostor placement determines several other parameters of the impostor, which are then used during the generation process. Each impostor is bound to a `border`, which is referred to as the “imposted border”. It is the same border on which the impostor is later placed during browsing.

3.2.1 Texture area

The first point of the generation process is to determine which area of the browser window displays geometry relevant to the impostor. This area will form the texture of the impostor, so it is always rectangular. The bottom edge is placed touching the imposted border.

Left-hand and right-hand edges are determined by an impostor parameter called `side_cutoff`. This parameter can specify none, one or both edges. Edges which are part of `side_cutoff` are truncated so that they touch the corresponding endpoint of the imposted border (see figure 7). Edges not thus specified are placed on edge of the entire browser window. Geometry which is in the view frustum but falls outside the texture area upon projection is not included in the impostor.

For impostors representing a view down a street, `side_cutoff` contains both edges, to minimize impostor size. Geometry present beyond these edges would be invisible anyway because of the houses constituting the street. Sometimes, however, an impostor represents a view which opens up on one or both sides (see figure 8, imposted border is highlighted). In such a case, even geometry

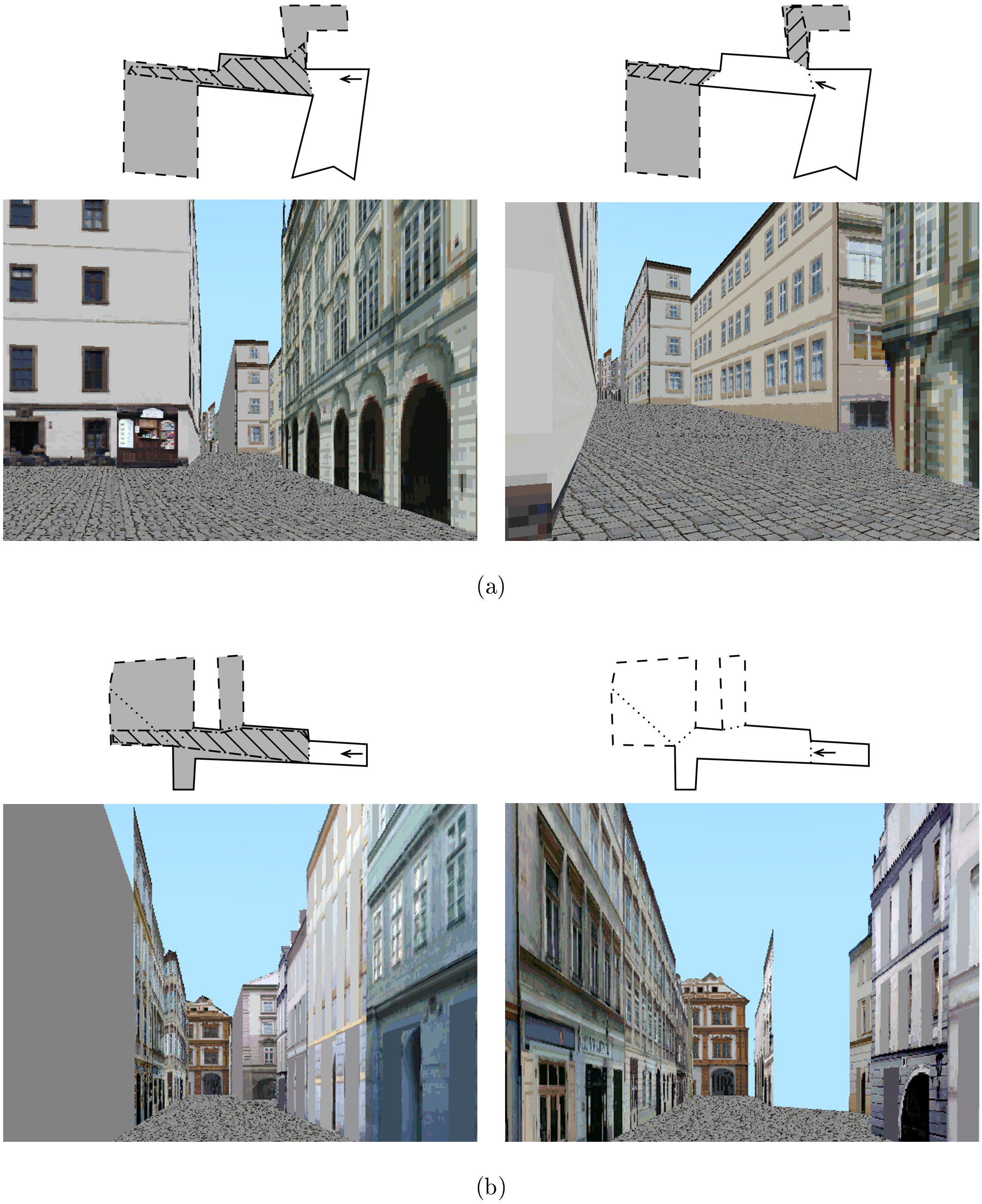
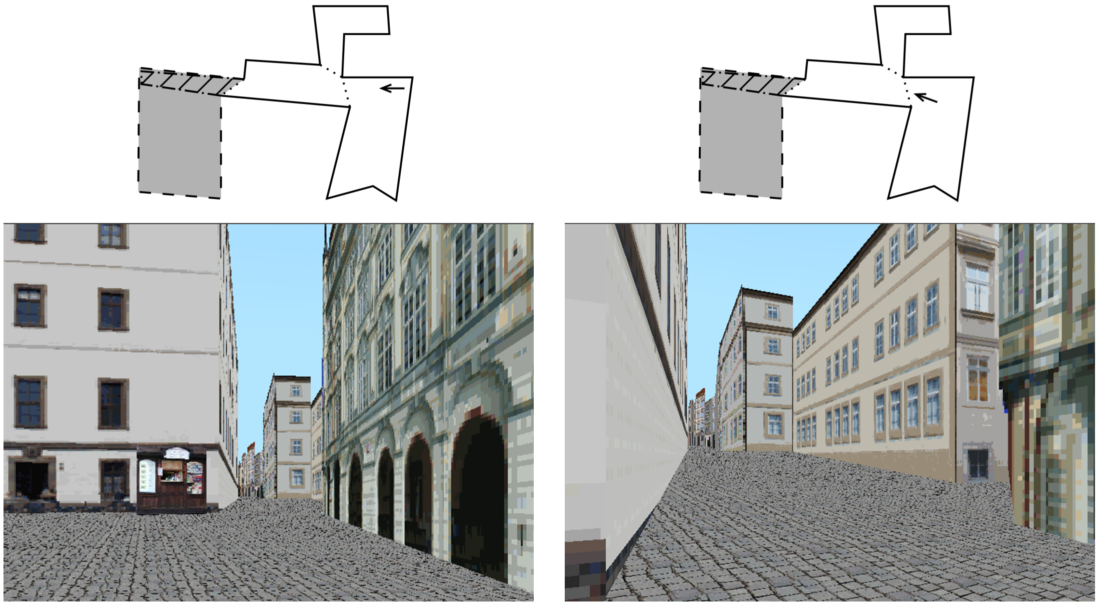
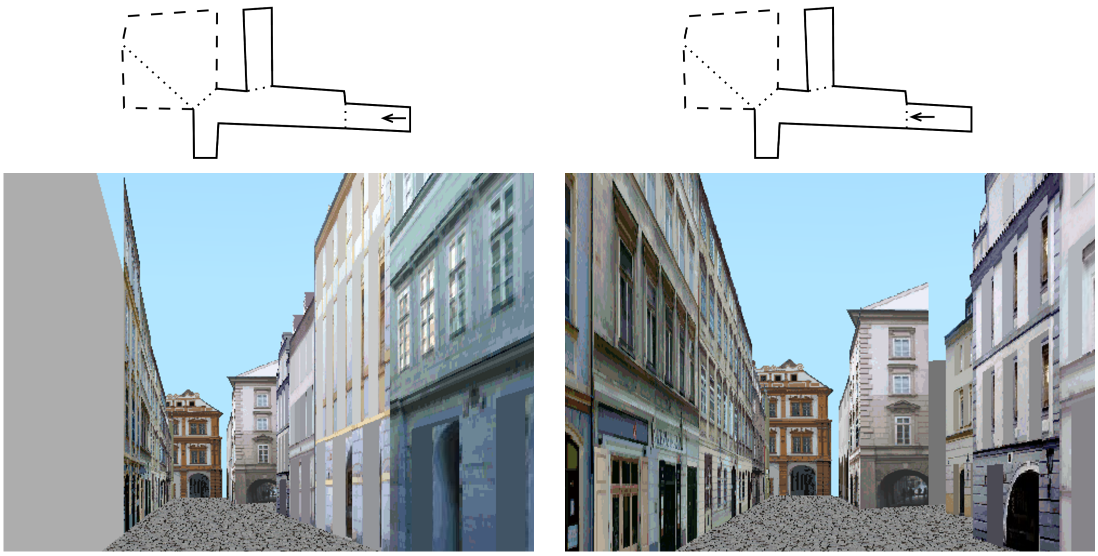


Figure 5: Limited visibility impostor regime



(a)



(b)

Figure 6: Edge-only impostor regime

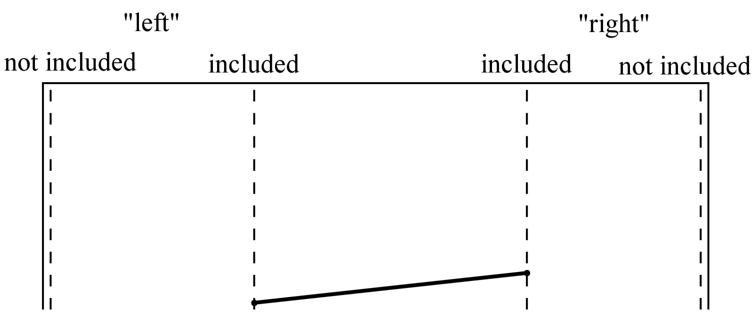


Figure 7: `side_cutoff` and texture area

beyond the imposted border (in figure 8, to the right) needs to be present in the impostor. Not including it would result in a wall present in the impostor where none is in the scene.

3.2.2 Border images

Due to reasons described in section 4.1, the generation process has no access to the Z-buffer or loaded geometry data. Thus, determining the original depth of a pixel in the texture is impossible. Yet this depth information is necessary for the creation of a depth-augmented impostor.

Fortunately, the structure of VOP allows for a simplification to be made which makes it possible to re-project a point in the texture back into 3D. From the point of view of VOP, a house is a flat vertical plane placed on a border[12]. Exceptions to this rule will be dealt with later. If a pixel is known to be part of the geometry of house h, it can be reprojected by assuming its original 3D position lies in house h's plane.

The problem remains of determining which house the pixel belongs to. VOP models can contain arbitrary geometry, so trying to retrieve geometry information from the files used would equal writing a new VRML browser (or at least VRML scene loader). A solution was found in displaying the scene in separate parts. Each house would be rendered separately and data about its image stored. In reality, it is not even necessary to do this on a per-house basis, as all houses (`lnodes`) on one border share the same plane. Therefore, pixels in the texture are assigned to borders. This is accomplished by capturing and processing the image of each border of each sector imposted by the impostor being generated. Which sectors these are is another parameter determined during impostor placement.

Not all borders in the imposted sectors are visible from the generation point. Some of them can fall outside the browser window, some are occluded by other geometry or viewed from the wrong direction and thus backface-culled (such borders are referred to as “inverted”). Borders in imposted sectors to be excluded from the generation process are also identified during impostor placement.

The entire captured image of a border is not necessary for the generation process. Instead, only the top and bottom contour are needed. The top contour is extracted from the image, while the bottom one is computed so that it lies directly on the line specified by the border’s starting and ending vertices. This contour is used to anchor the impostor on the ground, and retrieving it from the image would make ledges or other above-ground features extruding beyond the actual border line compromise its usability for this purpose.

There are exception to the rule which states that a house is a flat plane. The most common of these are house roofs. In the VOP, houses do not have proper roofs, but the front part of a roof, the one visible from the street, is usually part of the house model. Other exceptions exist too, though these are model-specific.

Shapes extruding out of the border plane present a serious problem. Figure 9 demonstrates what happens when a roof is incorrectly determined to lie within the plane. There is no solution to this problem that could preserve the offending shape’s 3D position without knowing something about the actual geometry. Fortunately, the most common case of such a shape are roofs, and these can safely be omitted from the impostor altogether, as their impact on visual quality is small. It can thus be sacrificed as part of the overall visual quality reduction brought by an impostor. This is achieved by truncating the sides of the border’s contour so that it cannot extrude past the border

[12]For the present purpose, indented windows and similar small extrusions can safely be ignored.

line itself (see figure 10).

The problem still persists for shapes which cannot be simply discarded, however. Several attempts were made to identify and correct such occurrences automatically, but none of them was successful. In the end, it was deemed necessary to obtain additional information about the geometry. If a designer wants an extruding part of a house to be preserved, they must provide information about the plane the extrusion lies in (or seems to lie in when viewed from the side), which is stored in the database. Such a shape is then treated as a fake border (i. e. having its own plane) in the generation process.

3.2.3 Vertices

Now that the planes and contours of all participating borders are known, the impostor geometry can be computed. All imposted sectors are displayed and the image stored to be used as a texture map for all faces of the impostor. The next task is to determine which pixels belong to which border.

A pixel can only belong to a border which includes it within the area defined by its contours. A border is assumed to contain no holes, that is, to occupy the entire space between its top and bottom contour. For each of these candidate borders, the pixel is reprojected into the border's plane. It is assumed to belong to the border for which the reprojected point is nearest to the camera[13].

With the information on the depth-ordering of borders, it is possible to establish positions of vertices comprising the impostor mesh. Vertices are placed on "points of interest". These are points which correspond to pixels on depth disparity lines. Points of interest are searched for on contours only, using the following criteria:

- A pixel where the border starts or stops being occluded
- A pixel where the border begins or ends
- A pixel where the border starts or stops occluding another border

The vertex mesh is then regularized to consist of quads. A vertex's texture coordinates are simply the screen coordinates of its corresponding pixel.

Unlike the method used in [5], there is no additional augmenting regular grid involved. Instead, the vertex mesh is refined locally as necessary. First, boundaries of the mesh are compared to the border's actual contour. If discrepancies above a certain threshold are encountered, the mesh is refined with more points to eliminate them.

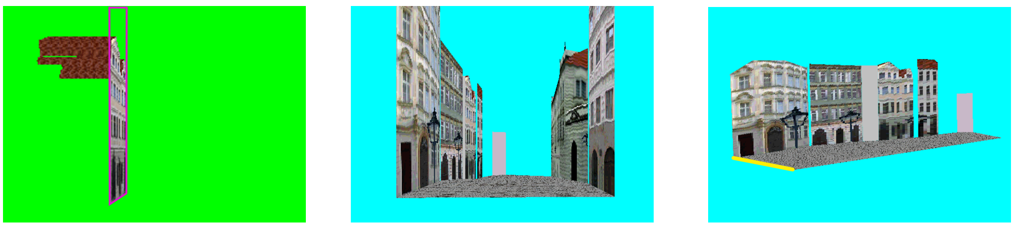
For borders perpendicular to the viewing direction, this is sufficient. However, borders more or less parallel to the viewing direction require further refinement, as they are most affected by perspective distortion. For such borders, the mesh can be refined by specifying an extra vertex in each quad's centre or subdividing the quad into four quads and refining these recursively. The criterion for the scale of this refinement is based on the border's angle to the view direction and its width in the texture. These together measure the border's impact on impostor visual quality and its susceptibility to perspective distortion.

[13]The process is not actually carried out for each pixel. See "Assigning pixels" in section 4.4.3 for details of the algorithm used.

Figure 8: A view which warrants non-full `side_cutoff`

Problematic roof (assumed plane highlighted)	Generated impostor	Impostor from the side (imposted border highlighted)

Figure 9: Roof wrongly assumed to lie in house's plane



Roof truncated (clipping region highlighted)	Generated impostor	Impostor from the side (imposted border highlighted)

Figure 10: Truncating the roof

3.2.4 Faces

In the previous step, vertex positions were computed so that each border consists entirely of quads, some of them possibly enhanced with a centre point. Thus, specifying faces for borders is trivial.

More faces are needed, however. A partially occluded border b needs to be connected by a face to the border which occludes b. Otherwise, there would be holes in the impostor (see figure 11). Such faces are called "connector faces".

If connector faces are textured like regular border faces, they appear like long horizontal stripes, seriously damaging visual quality. To avoid this, they are rendered as flat colour faces instead.

3.2.5 Floor

The most common floor texture in VOP is cobbling. It is a regular pattern, highly susceptible to aliasing. Other floor textures also lose detail rapidly with increased viewing distance. Using impostor texture on floor would result in an obvious visual difference between full-geometry floor and impostor floor. To avoid this, floor is not textured like the rest of the impostor. Instead, a single floor texture from the imposted sectors is chosen and used in full quality to texture impostor floor. As floor appearance is usually consistent in neighbouring sectors, the texture is likely to be already loaded, or will be needed anyway once the impostor is replaced by actual geometry. Thus, this approach does not present extra overhead.

Border vertices which are part of the bottom contour are included in the floor. Delaunay triangulation is then used to construct a triangle mesh for the floor.

3.3 External impostors

While the system is designed with automatically-generated impostors in mind, it does not rely on that fact. Any VRML file can be used as an impostor, as long as it is in the proper place in the filesystem and entered in the database correctly. This means hand-made impostors, possibly with real-world photography textures, can easily be added. This is useful for views of areas which are not part of the model, for instance. These would probably be included as surround-type impostors for sectors which contain the view. Note that surround-type impostors can be placed on any type of border.

The system assumes the following about the impostor file:

- It is located in the directory `impostors/<street_dir>/`
- Its name (without extension) is stored in the `gener_impostor` table, and its extension is '`.wrl`'.
- The origin of its coordinate system is the imposted border's right-hand vertex when looking at the impostor. If the imposted border is inverted (as is the case for surround-type impostors), this is its endpoint (B vertex). Otherwise (as normal for impostors placed on gates), it is the border's starting point (A vertex).

  The coordinate system is a rotation of the standard one around the Y axis. The X axis is parallel to the border's vector (when projected into the XZ plane) so that the vector $(-1, 0, 0)$ points from the origin to the border's other vertex.

  The Z axis is perpendicular to the X axis as normal for the basic VRML coordinate system. The Y axis points upwards. Figure 12 shows the coordinate system.

The scale of the impostor file is the same as that of the world. The impostor geometry is not scaled in any way before being inserted into the scene.

- To display the impostor, it must be entered either into the `surround_impostor` or `proxi_impostor` table. For impostors placed on gates, sectors imposted by the impostor must be entered into the `imposted_sectors` table.
- If the imposted border is inverted, it must be listed as such in the `gener_impostor` table.

3.4 Impostor implementation

This section describes the implementation details of impostor displaying, as it was outlined in section 3.1. Parts of the impostor generation process which affect the existing VOP are also described here. For implementation details of the generation program itself, refer to section 4.4.3.

3.4.1 Changes to the database

All information needed for impostor generation and display was added as new tables into the database. These tables are described below.

`gener_impostor (border_id, imp_name, place_type, x, y, z, orient, pitch, fov, inverted_borders, hidden_borders, side_cutoff, surface_appearance)`

This table is primarily used during impostor generation. The only field used during displaying is `imp_name`, which contains the filename of the impostor's file, without extension ('`.wrl`' is assumed).

The rest of the table contains data for generating the impostor. `place_type` determines whether the position was generated by the impostor placement algorithm or entered by hand. `x`, `y` and `z` determine the generation point. `orient` specifies camera rotation around the Y axis in radians, with 0 pointing due north and positive values rotating counter-clockwise. `pitch` is rotation around the X axis, where 0 means level and positive numbers rotate towards the floor. `side_cutoff` specifies which edges of the impostor can be truncated to align with the imposted border. `surface_appearance` is a reference into table `appearance` and identifies the appearance to use for the impostor's floor. A value of -1 indicates the impostor has no floor.

`inverted_borders` is a list of border IDs (references to table `border`), which are seen from their right-hand side. `hidden_borders` contains a list of IDs of borders which should not be used during the impostor generation at all. Both of these fields are given as a sequence of numbers separated by arbitrary amounts of whitespace and/or commas.

`imposted_sectors (impostor_id, sector_id)`

This table stores which sectors are imposted by each impostor. It contains references into tables `gener_impostor` and `sector`.

`proxi_impostor (border_id, impostor_id, koef)`

This table defines impostors present in the world. `border_id` is a reference into table `border`. `impostor_id` (a reference into `gener_impostor`) specifies the impostor present on this border.

`koef`[14] defines a scaling factor to be applied to the distance at which this impostor is hidden and replaced by actual geometry in the limited visibility impostor regime (see section 3.1.2).

3.4.2 Changes to script `vrml.php`

The entire algorithm managing impostors is implemented in the main `SCR` script. It introduces two states for a loaded sector: it can be either "revealed" (visible) or "imposted" (not visible). Similarly, an impostor can be either "shown" or "hidden".

The goal of the algorithm is to ensure an impostor and a sector it imposts are never visible at the same time. This is accomplished by the following rule: always display an impostor in preference to the sector, unless the sector is marked as not impostable. Both impostor management regimes (see section 3.1.2) are achieved using this rule. The difference lies in which sectors are marked as not impostable. This information was added to the parameters of sector switching, which is dealt with in the script `surround.php` (see section 3.4.3). There are a few other differences between the two regimes; these will be discussed as necessary.

Several associative arrays had to be added to `SCR`. For arrays which are never iterated over, these were implemented simply using stringified keys in the appropriate `MF*` type. Where this didn't suffice, an approach similar to that already present in VOP was taken[15]. The associative array (the variable prefixed with '`i_`') contains indices into the linear array, incremented by 1. The linear array is then prefixed with '`a_`' and contains actual data.

The following arrays were added:

`MFInt32 isNotImpostable` This is actually a boolean array, as it only ever holds values of 0 or 1. It is associative, indexed by sector ID. A value of 1 for sector *sect* means that *sect* is currently marked as not impostable. A value of 0 or no value at all means *sect* can be imposted.

`MFInt32 isImposted` This array is also actually boolean. It is associative, indexed by sector ID. A value of 1 indicates the given sector is currently imposted. This array can contain values even for sectors which are currently neither loaded nor being loaded. In such a case, the sector would be loaded as imposted.

`MFInt32 impostorShown` Another associative boolean array. It is indexed by impostor ID. A value of 1 indicates the impostor is currently shown, 0 or no value mean it is either hidden or not part of the currently loaded scene at all.

`MFInt32 visibleImpostors` This linear array holds IDs of all impostors which are currently shown.

`MFInt32 i_impostor` **and** `MFInt32 a_impostor` These constitute a linear/associative array pair. `i_impostor` is an associative array indexed by impostor ID and contains indices into the linear array `a_impostor`. `a_impostor` holds data about all loaded impostors. Each impostor occupies three elements in the array. If the record for impostor *impId* starts at index *idx*, the following holds:

- `a_impostor[`*idx*`]` contains *impId*
- `a_impostor[`*idx*`+1]` contains ID of the sector which contains impostor *impId*

[14] This stands for 'coefficient'. However, 'k' was used for consistency with original VOP tables `lnode` and `node`.

[15] See [1], section 3.2.3, for details of this system.

- `a_impostor[`*idx*`+2]` contains ID of the sector to which the gate holding impostor *impId* leads, the impostor's "destination sector"
- `i_impostor[cnv(`*impId*`)]` contains $idx+1$

Use of these arrays will be discussed in the relevant parts below.

The basic structure of sector management (`load`, `done`, `del`) has been left the same, but some of these steps were diversified to work differently for revealed and imposted sectors. The new control flow of sector management is illustrated in figure 13.

`load_sect` now calls `done_loading_sect` once the sector is loaded or retrieved from cache. `done_loading_sect` inserts all of the sector's impostors into `i/a_impostor` and calls either `done_imposted_sect` or `done_revealed_sect` as per the sector's current state. Finally, the function `signalRevelation` is called (see below).

`done_imposted_sect` calls the original `done_sect` and then sets the sector's `set_imposted` to true, which makes the sector invisible but doesn't remove it from the world.

`done_revealed_sect` also calls the original `done_sect`. Then, it marks the sector as revealed by calling `done_revealing_sect`. Here, all impostors of the added sector are processed by `try_adding_impostor`. A set of candidate sectors is established for the impostor. For the limited visibility regime, this set contains only the impostor's destination sector. For the edge-only regime, it contains all sectors imposted by the impostor; this information is retrieved from the `full_impostors` field of the sector being added. If any sector of this set is marked as not impostable, the impostor is hidden. Otherwise, the impostor's destination sector is imposted[16]. Finally, the system hides all impostors which have this sector as their destination.

`del_sect` has also been modified. It first clears the sector's not impostable mark (if any) and then hides all its impostors and removes them from the relevant arrays. The sector is then removed from the world and is restored into its original state (i. e. not imposted, all impostors shown) before being inserted into the cache.

`done_par` had to be modified too. Two new arrays were added to the sector-switching parameters — '`visible`', a list of sectors to be marked as not impostable, and '`impostable`', a list of sectors to have that mark removed. Before sectors are loaded or deleted, `isNotImpostable` is updated using information from these arrays. After loading and deleting sectors, these arrays are used to alter sector visibility. All sectors in `visible` are revealed and all sectors in `impostable` are imposted. Finally, visibility is updated by calling `signalRevelation`.

The code for showing or hiding an impostor and imposting or revealing a sector is rather complex, as these operations can result in a cascade of other impostors being shown or hidden and sectors being imposted or revealed. For example, assume impostors i_1 and i_2 both impost sector *sec*. Hiding i_1 then results in revealing *sec*, which in turn means i_2 has to be hidden.

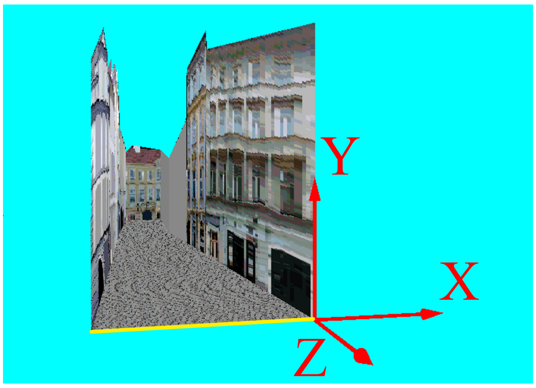
Impostor and sector visibility changes when an impostor is added to the scene (it is shown or part of a sector being added or revealed) or removed from it (it is hidden or part of a sector being removed or imposted). When an impostor is thus added, it is processed by `try_adding_impostor` (detailed above) which results in either hiding the impostor or imposting its destination sector. When an impostor is removed, its destination sector is revealed.

Imposting a sector removes all its impostors from the scene and shows all impostors imposting it. Revealing a sector calls `done_revealing_sect`, described above.

[16] In the edge-only regime, this can only happen if none of the candidate sectors is actually loaded, including the destination sector.

View from near generation point Different angle reveals holes Connector faces added

Figure 11: Connector faces



(imposted border is highlighted)

Figure 12: Coordinate system of impostor file

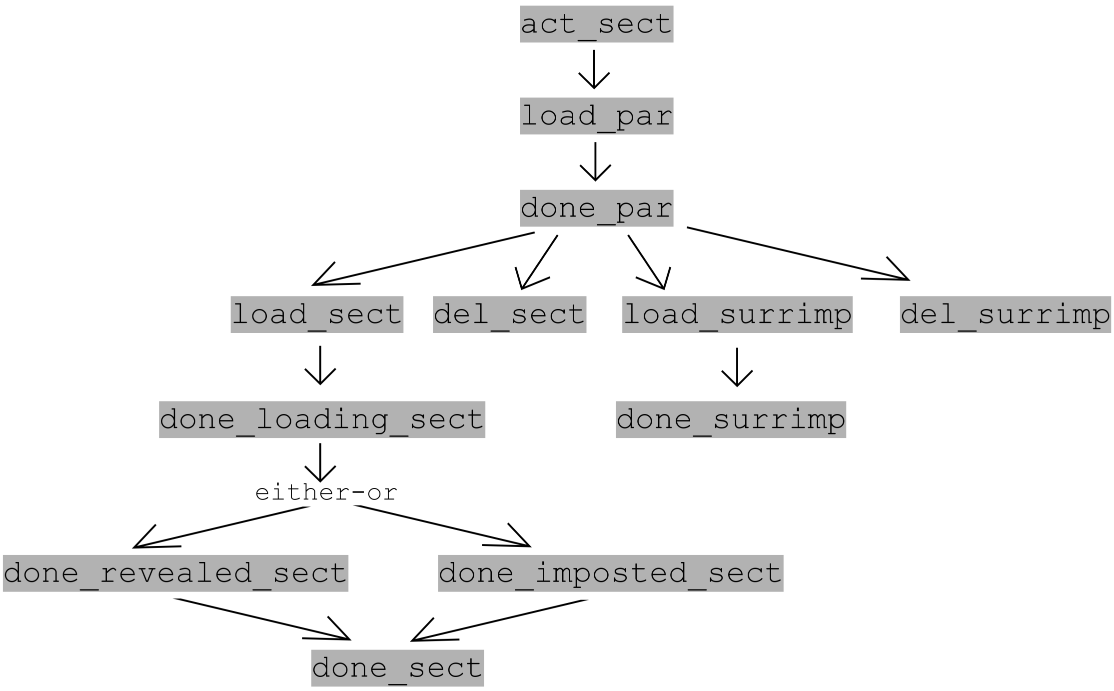


Figure 13: Sector management function dependencies

In the limited visibility regime, an impostor can also be removed or added when the user comes close enough to it or walks away from it, respectively.

Dependencies of all the functions managing impostor and sector visibility are shown in figure 14. Bold arrows indicate functions callable from outside the graph.

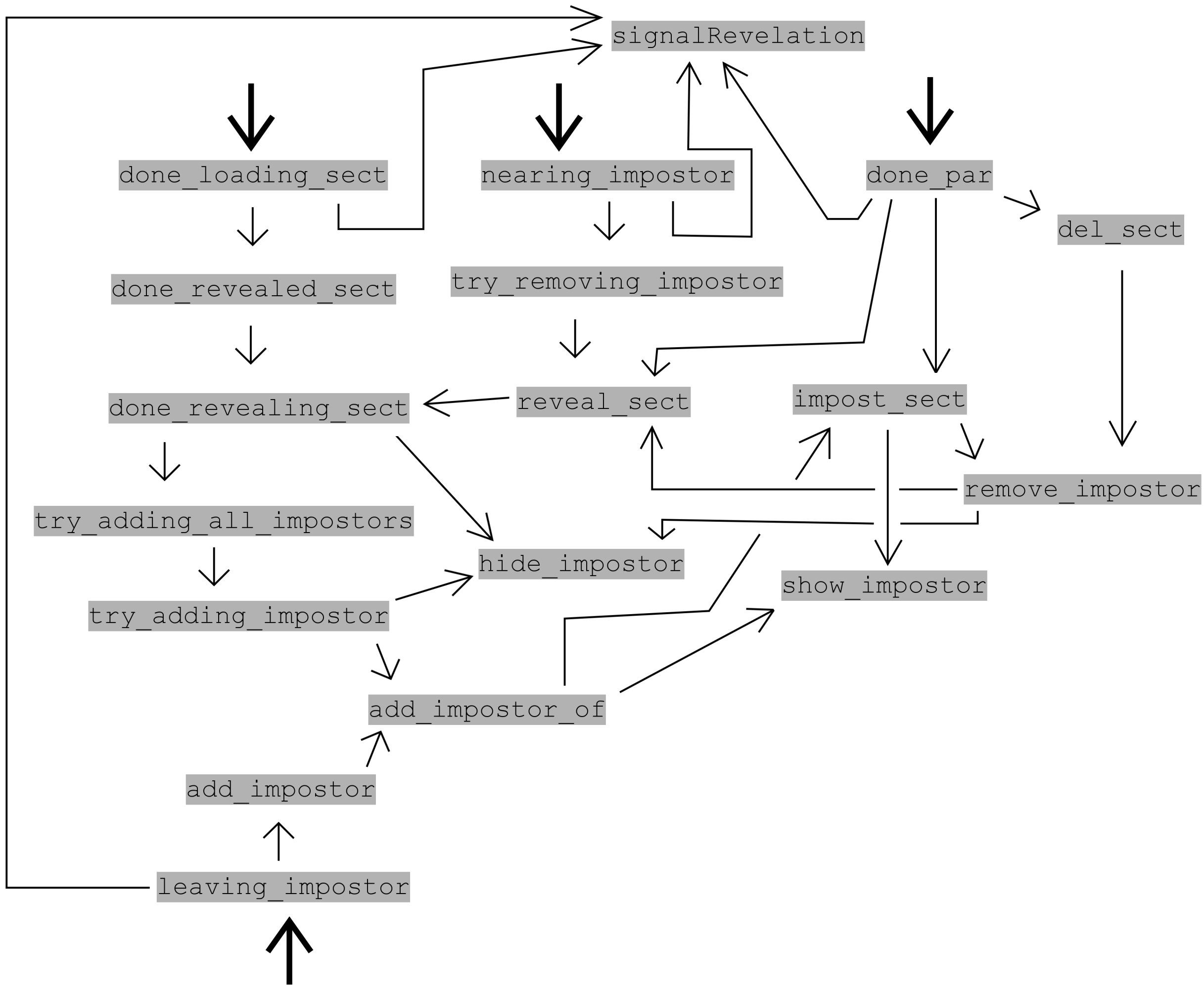


Figure 14: Impostor management function dependencies

Matters are complicated further by VRML's constraint of one event value sent per time-stamp. This led to the necessity of creating the array `visibleImpostors` and adding the function `signalRevelation`. Basically, all sector and impostor visibility changes are only carried out in the data structures of `SCR`. When the current cascade processing is finished, `signalRevelation` is called. This traverses all loaded sectors and sends each of them the current contents of the array `visibleImpostors` as a `set_visibleImpostors` event. This causes the sectors to actually show or hide their impostors as required.

Originally, impostors were hidden immediately when a sector was revealed. However, this led to occurrence of the following situations: Sector *sec* is to be loaded and revealed, due to the user walking to a new sector (perhaps one adjacent to *sec*). Impostors imposting *sec* are hidden, but *sec* is not loaded yet. This means that the impostors vanish, leaving a hole in the model until *sec* is loaded.

This behaviour was the result of trying to prevent the user from coming too close to an impostor, as an impostor's visual quality at close-up is poor. However, it is still better than nothing at all. The mechanism was therefore modified so that impostors imposting a sector are only hidden when the sector is added to the world (i. e. as part of `done_loading_sect`). The algorithm description above already describes this final state.

Changes to gates The limited visibility impostor regime requires support in the `PROXG` prototype. In this regime, an impostor is hidden when the user comes within a certain distance of it (this distance can be specified in session configuration). The idea of the implementation is the following: a proximity sensor, `IMP_NEAR`, of the specified size is added to the gate. The gate also holds information which impostor it contains. When the user enters or leaves `IMP_NEAR`, the gate sends an appropriate event (`hide_impostor` and `show_impostor`, respectively) containing its impostor ID. These events are routed to `SCR` (to its `nearing_impostor` and `leaving_impostor` events), which removes or adds the impostor. This follows the general scheme of gate–`SCR` communication as already present in VOP.

Implementing the idea exactly as presented above didn't work. The problem is that gates are actually re-used when the user enters a new sector. That is, a sector doesn't have its own instances of gates, but there are several global instances and these are just repositioned onto the new sector's `proxi` borders. This had the following unfortunate consequence: when passing through a gate, the user could remain in the area of its `IMP_NEAR`, as the `PROXG` was just spun around to represent the new sector's gate. Thus, no relevant events were sent by the proximity sensor and the mechanism would malfunction.

To prevent this, two such proximity sensors were added to `PROXG`. They are encapsulated in a `Switch` node and each time the gate is re-used, it switches from one of these sensors to the other, so that at most one of them is active at any one time[17].

The distance to which an impostor can be approached is scaled by a scaling factor '`koef`' on a per-impostor basis. This factor is transmitted to the gate as part of the `par` event, the type of which was changed from `SFVec2f` to `SFVec3f` to accommodate this.

Calibration scene The program requires a calibration scene for estimating projection parameters (see section 4.4.2). The scene consists of a cube of two metres a side, centred at $(0, 0, -11)$. It is included in the scene only under the `impgen` layout and is encapsulated in a `DEF CALIBRATOR Switch` to only be shown when needed. `CALIBRATOR` is controlled by an event `set_calibrationScene` in `SCR`.

Background The screen capturing process relies on changing the background colour to separate geometry from the background (see section 4.4.3 for details). Two contrasting background colours are alternated: bright green (RGB 0,1,0) and purple (RGB 1,0,1). Background of such colour is used in the `impgen` layout.

`eventIn SFBool switchBgrColor` has been added to `SCR`, which just routes it to the same event added to the `GLOOM` node. If `false` is sent to the event, the background colour is swapped from green to purple or vice versa. Sending `true` causes the colour to revert to green.

[17]None of them is active if the gate doesn't contain an impostor, or the edge-only regime is used.

Avatar position eventIn MFFloat jump_avatar was added to SCR to allow the applet to set the camera position. It packs several pieces of information into one event, so an array type was chosen. The array always has 9 elements.

The first three elements are the new position and the next four specify camera orientation (in normal VRML axis-angle format). The next element specifies field of view, while the last one holds ID of the destination sector. Either of these two can have the value 0, meaning present value should be used.

Blackouts and guidance lines Several events were added to SCR which the applet uses to modify visibility of parts of the scene. SCR processes them simply by forwarding them to the relevant sectors.

3.4.3 Changes to script surround.php

Two MFInt32 fields were added to the sector-switching parameters[18]. visible contains a list of sectors which must be displayed (i. e. they must not be imposted) when *dest* is entered. impostable is a list of sectors which can safely be imposted. Which sectors go where depends on the impostor management regime used. Either way, it is ensured these arrays only contain the sectors which are visible (as per the visibility table) from *dest*.

For the edge-only regime, all sectors go into the visible field and impostable is empty. This means that no loaded sector can be imposted. Thus, impostors can only be displayed for sectors which are not in the loaded part of the world — precisely what this regime needs.

The computation for the limited visibility regime is slightly more complicated. A set of all sectors imposted by all impostors in *dest* is retrieved from the database. Any of the sectors visible from *dest* which fall into this set are put into the impostable array. The rest of the sectors visible from *dest* goes into visible. This means that in this regime, no sector is prevented from being imposted if an impostor exists for it.

3.4.4 Changes to script sector.php

There are two kinds of changes to sector.php: displaying impostors and support for impostor generation. However, both of these require events to be transmitted from the world-controlling SCR script to sectors. This communication is routed through the sector's MY_PAR node, which has been enhanced for this purpose by many event handlers.

To accommodate for all the changes necessary, a sector's VRML structure was modified as shown in figure 15.

A disadvantage of MY_PAR is that it has to be the first child of the sector's top-level Group node (this is relied upon in the main SCR script). This means it cannot have direct access to any nodes in the sector, as it comes before they have a chance of being DEFed. Thus, another script node, called "GUIDANCE_SCRIPT", was added to the very end of the file, and it performs operations where direct access to nodes is necessary. MY_PAR's script and GUIDANCE_SCRIPT communicate with each other via VRML routes.

Two arrays were also added to MY_PAR. The first one, MFInt32 impostors, is always present. It contains two elements for each impostor present in the sector. The first element is impostor ID, the second is ID of the impostor's destination sector. The other array, MFInt32 full_impostors,

[18] This section assumes parameters for switching from *src* to *dest* are being established.

```
Group {
 DEF MY_PAR PAR { ... }
 DirectionalLight { ... } #if sector has light specified
 DEF GEOMETRY_ROOT Switch {
  Group {} #empty
  Group {
   DEF BORDERid Switch { #one for each border
    Group{} #empty
    Transform {
     DEF IMPOSTORid Impostor { ... } #if border contains impostor
     #nodes for border's geometry
    }
   }
   DEF SECTOR_CONTENTS Switch {
    Group {} #empty
    Group {
     #nodes for free-standing node's geometry
     #nodes for floor and/or ceiling
    }
   }
  }
 }
 DEF BORDER_GLid GuidanceLine { ... } #one for each border; only if enabled
 DEF LNODE_GLid GuidanceLine { ... } #one for each lnode; only if enabled
 DEF GUIDANCE_SCRIPT Script { ... }
}
```

Figure 15: Structure of sector VRML file

is only present in the edge-only impostor regime. It too contains a record for each impostor in the sector, but these records have a variable number of elements. The first element of each record is impostor ID and subsequent elements are IDs of all sectors imposted by the impostor. The last element of each record is -1. Basically, it is a two-dimensional array linearized into one dimension. `SCR` uses this array when checking whether a sector imposted by an impostor is marked as not impostable.

Displaying impostors A new prototype, '`Impostor`', was created for adding impostors to the scene. It is basically an `Inline` node wrapped in a `Switch` which allows it to be removed from view. It supports some additional debugging functionality.

When a sector is imposted, it has to be removed from view while still part of the scene graph. To facilitate this, the entire geometry of the sector has been enclosed in a `Switch` node called `GEOMETRY_ROOT`, with an empty `Group` as an alternative. This is controlled by `MY_PAR`'s `set_imposted` event handler.

When an impostor should be shown or hidden, each sector is informed by receiving a `set_visibleImpostors` event containing an array of all currently visible impostors. `MY_PAR` delegates this event to `GUIDANCE_SCRIPT`. `GUIDANCE_SCRIPT` has a list of impostor nodes in this sector and their IDs in arrays `MFNode impostors` and `MFInt32 impostorIds` respectively. Using this information, it alters the visibility state of all the sector's impostors accordingly.

Impostor generation The impostor generation process places special requirements on the sector geometry. Here, their VRML implementation is described. For details of when and how they are used, refer to chapter 4.

Complete blackout of the sector is performed by the `set_blackout` event, which is delegated to `GUIDANCE_SCRIPT`. It is implemented using `SECTOR_CONTENTS` and the `BORDER`*id* nodes of all borders. Using these rather than the all-encompassing `GEOMETRY_ROOT` allows blacking in of individual borders in a blacked out sector. This is controlled by the given border's `BORDER`*id* node from within `GUIDANCE_SCRIPT`'s `set_border` event handler.

Displaying of a border's guidance line is controlled by the `set_borderGuidanceLine` event, again delegated to `GUIDANCE_SCRIPT`. It is implemented using the line's `BORDER_GL`*id* node, an instance of the `GuidanceLine` prototype. This prototype is a simple wrapper for an `IndexedLineSet` tailored for displaying a single line and properties to set its visibility and colour.

Parameters of both `set_border` and `set_borderGuidanceLine` are identical. It is an integer array of two elements. The first element is the ID of the border, and the second one is either 1 (for visible) or 0 (for hidden). These events are implemented using the arrays `MFNode borders`, `MFNode borderGLs` and `MFInt32 borderIds`.

Note that guidance lines (both for borders and `lnodes`) are only present in the scene graph during impostor generation (as determined by page layout) and not during normal browsing.

Legacy code At one time during development, each `lnode` had its own guidance line. This concept was later abandoned, but the code for displaying these guidance lines was left in place for potential future use. The controlling event, `set_lnodeGuidanceLine`, expects a four-element array as its parameter. The first element is `lnode` ID and the other three elements correspond to red, green and blue values (in a range of 0–255). Setting all three to 0 hides the guidance line, while any other value makes it visible and sets its colour accordingly.

3.4.5 New script: `saveimp.php`

When an impostor is generated, the program uploads it to the server by connecting to this script. The script takes two parameters, `id` and `what`. `id` is the database ID of the impostor to be uploaded. `what` is a string, containing either '`image`' or '`geometry`'.

The script retrieves the correct filesystem position for the impostor from the database. It then reads data from its standard input and saves it in that position. Impostors are placed in the subdirectory `impostors/<street_dir>/`. An impostor's filename is stored in table `gener_impostor`.

The script's parameter `what` determines the content of the data being uploaded. '`image`' specifies the texture, which is saved with the extension '`.png`'. '`geometry`' means the data contains the impostor's VRML file, which will be saved with the extension '`.wrl`'. No other values of `what` are valid.

If the script executes successfully, it writes the value 0 (as a byte) to its standard output. If its `what` parameter was invalid, it writes 1. Inability to save the data is signalled by 2.

4 The program

This chapter describes the Java program which performs impostor generation and impostor placement.

4.1 Why an applet *and* an application?

The program consists of two parts — an applet and a stand-alone application. This rather unorthodox state is a result of a combination of several unfavourable circumstances.

The chosen method of impostor generation, based on [5], requires access to the Z-buffer and the ability to render the scene into a texture. This is of course not possible with a VRML browser plugin of a web browser, as from a programmer's point of view, this functions as a 'black box', with no means of accessing its internal data.

An attempt to use an open-source VRML browser, Xj3D[19], was made. Considerable time was spent configuring Xj3D and modifying VOP for display in it. In the end, a bug in Xj3D's ECMAScript interpreter was found which renders it utterly incapable of running the scripts needed for displaying VOP, so Xj3D had to be abandoned. Other open-source browsers were briefly considered, but none of them seemed as complete as Xj3D, so it was unlikely any of them would be capable of displaying VOP in its complexity. Thus, a different approach was chosen.

It was decided to create a Java applet that would communicate with the VRML browser displaying VOP using the external authoring interface, EAI. Image data would be acquired by capturing the rendered scene directly from screen. The applet would process the data and communicate with the database as necessary.

Available VRML browser plugins were analyzed. Of those available for free, supported and EAI-capable, only Cortona VRML Client from ParallelGraphics[20] was capable of reliably displaying VOP. However, Cortona's EAI only works on Microsoft implementation of the Java virtual machine, which is only available in Microsoft Internet Explorer. Cortona itself works in other internet browsers as well, but other Java virtual machines cannot display EAI applets due to missing libraries[21].

This complicated matters further as the Microsoft virtual machine only supports Java version 1.1, which does not include the screen-capturing capability. Thus, it was decided to split the program into two parts, an applet that would run in the browser on Microsoft Java, and a stand-alone application that would run on an up-to-date virtual machine. It was expected to use RMI[22] for communication between them. However, Microsoft Java doesn't support RMI, so it was settled on the applet and application communicating via a TCP socket.

It requires a signed applet to be able to open a TCP socket to a host other than the one which the applet comes from (this includes `localhost`). Unfortunately, Microsoft Internet Explorer does not honour signing policies of Sun Java (signed `JAR` files) and requires its own version, signed `CAB` files. These can no longer be created, as Microsoft has dropped support for its Java SDK which included the necessary tools.

The only solution left was to have the applet loaded locally. This requires running a PHP-capable HTTP server on the machine running the program. VOP configuration menu has been

[19] Available from `http://www.xj3d.org/`

[20] Available at `http://www.parallelgraphics.com`

[21] This is true for both the old and new version of EAI.

[22] Remote Method Invocation, a native Java way of communication between two programs running in different virtual machines [7].

modified to include an option to load the applet frame from a user-specified path on `http://localhost` instead of from the VOP server. The applet frame file, `impgen.php`, is provided, it just needs to be placed somewhere in the local server's document directory, along with `impgen.jar`. This results in the applet originating from the local machine and thus able to open sockets to it normally.

While inconvenient, the requirement for running an HTTP server locally is not crippling, as the program is only run during world-design time by the designers, never by users just browsing the town. An alternative to installing an HTTP server is to run the program directly on the machine running the VOP server, if such an option is available.

Note that these restrictions are only imposed by Cortona being the only usable VRML plugin. Neither the applet, nor the application relies on Microsoft Java or the specifics of Cortona in any way. Thus, if a VRML plugin was found which can display VOP and works on Sun Java, the applet could easily be modified and deployed in a signed `JAR`, thereby allowing it to communicate with the application even if running on a remote server. This would remove the above mentioned requirement of local server installation altogether.

4.2 User interface

The program is controlled entirely by the applet. The application window only displays progress information, so it can safely be minimized while running. Note that for the program to function correctly, the VRML browser window must not be obscured by anything, as it is processed by screen capturing.

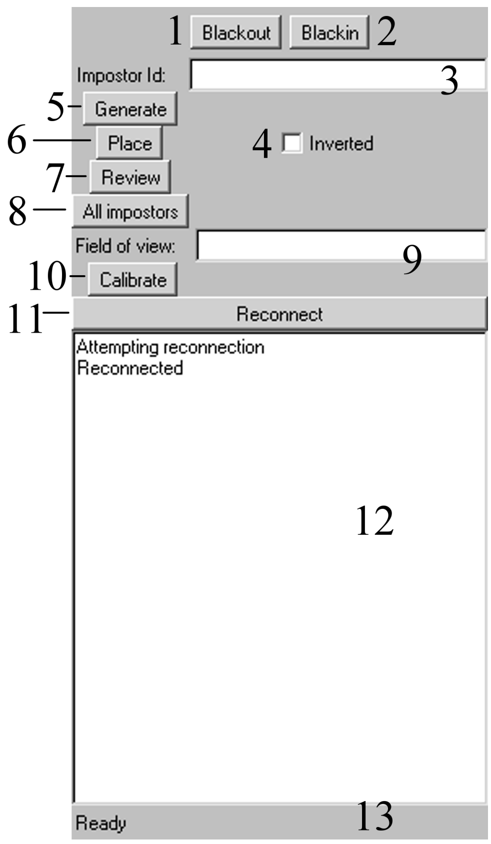


Figure 16: Applet interface

The applet's interface is shown in figure 16. Error messages and work progress messages are displayed in area (12). The status bar (13) displays the applet's current state. If the applet is currently performing an operation, the text "Busy..." is displayed and all controls are disabled.

Buttons (1) & (2) control scene visibility. Pressing button (1) causes complete blackout — all sectors are hidden from view. Button (2) is the reverse — it displays all sectors. These are seldom needed, as blacking out is carried out automatically whenever necessary.

The main tasks of the applet — impostor generation and placement — are carried out using the buttons (5), (6) & (7). These initiate the appropriate operation on any impostors listed in field (3). One or more impostor IDs (database IDs of borders) can be entered, separated by whitespace, commas or periods. Pressing button (8) fills in the field with all impostors currently existing in the database.

Button (5) starts the impostor generation process. Impostors entered are processed sequentially, any invalid IDs are skipped. Each impostor generated is uploaded to the VOP server.

Button (6) initiates impostor placement on the borders specified. If (4) is checked, the borders are treated as inverted. Use this option when placing a surround-type impostor which will be looked at from outside the border's sector (see figure 17). The placement process first determines the generation point and `side_cutoff` and then computes imposted sectors, inverted and hidden borders.

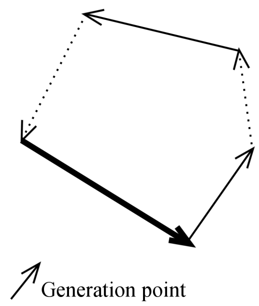


Figure 17: Inverted border for surround-type impostor

Determining the generation point well relies on understanding the world topology and requires a degree of aesthetic feeling. Therefore, the automated placement system can sometimes fail at this task, choosing a clearly unsuitable location for the generation point. At other times, the designer can want the impostor seen from a particular point because of its interaction with the rest of the scene, for example. In such a case, it is possible to enter the generation point into the database by hand. Then, button (7) can be used to perform only the second half of the placement process. It starts from the generation point in the database and only computes the imposted sectors and inverted/hidden borders. It is also affected by the checkbox (4).

Both the generation and placement process rely on knowing the projection parameters of the VRML browser. These are computed automatically when necessary by a process called "calibration" and stored in a file. It is also possible to start calibration by hand, using button (10). This starts the calibration process for the field of view specified in (9). The number entered is the arc of view, in radians. The parameters determined by the calibration are then saved into the file as normal (see section 4.4.2 for details).

Button (11) is used to re-establish the TCP socket used for communicating with the stand-alone application. Pressing it closes an existing socket, if any, and tries to open a new one. It can be used if the applet is started before the application and is thus unable to connect at startup, or if the application had to be restarted for whatever reason.

4.3 Inside the applet

This section briefly describes important implementation details of the applet.

4.3.1 External authoring interface

The core of the applet is the EAI access to the VRML browser. It is implemented using the "new" EAI [3]. Presently, the applet only sends data into the VRML scene, but at earlier stages of development, it relied on reading data from the scene, too, so the new EAI was used.

The applet only accesses the node `SCR`, namely its following events:

`switchBgrColor`, `set_blackout`, `jump_avatar`, `set_borderGuidanceLine`, `set_lnodeGuidanceLine`[23], `set_border`, `blackinSectors`, `blackoutSectors`, `set_calibrationScene`.

There are many occasions when the algorithm has to wait for a rendering operation to complete before proceeding (usually before capturing the rendered image). Unfortunately, there is no way of programmatically telling when the VRML browser has finished rendering. Thus, this was solved by waiting for a fixed timeout after such an operation is requested. The length of this timeout is specified as an applet parameter.

4.3.2 Parameters

The applet uses the following applet parameters (specified by the `<param>` tag in HTML). If any of these is missing, the applet will use a default value and output a warning.

`impgenport` This parameter specifies the port number for the socket for communicating with the application. The default value is 2404.

`socketretries` Specifies the number of attempts to open the socket before failure is reported. Default is 1.

`eairetries` Specifies the number of attempts to connect to the VRML browser before failure is reported and the applet stopped. Default is 5. For similarity to the existing VOP applet, `BMapEAI`, this parameter can also be named `maxtrials`.

`vrmlframe` Name of the HTML frame in which the VRML browser resides. Default value is an empty string, which means the applet and the browser are in the same frame.

`vrmlxoffset`, `vrmlyoffset` These parameters specify the position of the VRML browser window's top left-hand corner. It is given in pixels, relative to the top left-hand corner of the applet. Together with its dimensions, below, these parameters determine the area on screen to be captured as VRML output. Default values are 0.

[23]This event is presently unused.

vrmlwidth, vrmlheight These parameters specify the dimensions of the VRML browser's window. Default values are the width and height of the applet itself, respectively.

avatarheight This specifies the height of the avatar used in the VRML scene (as given in a `NavigationInfo` VRML node). It is used for placing the generation point. Default value is 1.75.

vrmldelay This parameter specifies the timeout for waiting for a VRML operation, in milliseconds. It depends on computer speed and rendering capabilities, and can be fine-tuned experimentally. The default value is 500, which is acceptable for a high-speed machine.

4.3.3 Threads and the socket

The applet runs in two threads. One is the AWT thread managing user interface. To prevent the applet from locking up while an operation is in progress (as this can take minutes if multiple impostors are being generated), the EAI calls and socket communication are processed in a different thread.

The threads communicate via a mechanism called `TaskHolder`. It represents a conditional variable on which the working thread waits until the AWT thread inserts a task (corresponding to the user clicking a button). The task is then retrieved by the working thread and the user interface is disabled to prevent the user from interrupting the task in progress[24]. Once all operations are finished, the interface is restored.

The applet is always the one initiating communication across the socket, so it does not have to listen on it permanently, only when expecting data from the application.

4.4 The stand-alone application

The application works like a reactive server. It listens on the socket and when the applet sends a request, it processes it. During such a processing, control can be transferred back and forth between the applet and the application, but they seldom work simultaneously. Basically, the entire program performs a sequential processing, with parts being executed in the applet and parts in the application. Still, it has been endeavoured to utilise time the applet spends waiting in a VRML timeout for computations in the application.

4.4.1 Parameters

The application requires a number of arguments to be specified, like the URL of the VOP server, database access information etc. The processes it computes contain a number of parameters (threshold values and the like) which can also be specified by the user. There are two ways of supplying arguments to the application.

The first is to list them on the command line when starting the application. A full list of arguments which can be specified in this way is obtained by invoking the application with an argument "`-help`". For arguments which take a value, the value must follow the argument, separated by a space (or as normal for a command-line argument).

Another way is to specify parameters in a property file. A property file has the normal syntax of Java property files; that is, each line contains a key–value pair separated by an equals sign '`=`',

[24] If, for whatever reason, the user needs to abort an operation, they can do it by terminating the stand-alone application.

a colon sign ‘`:`’, or whitespace. Some parameters can only be specified via property files, not on the command line.

The application always looks for a property file called “`ImpostorProgram.properties`” in the current directory. Additional property files can be specified via the command line argument “`-properties` *`file_name`*”. If a property is defined in multiple files, the last definition encountered is used. Note that command-line arguments, if specified, take precedence over values from property files.

A sample property file is supplied with the application. It contains a description of all properties together with their default values.

4.4.2 Calibration

Impostor generation and placement require computing projections of world points into the texture plane and re-projecting texture points back into 3D. For this, the projection matrix needs to be known. For VRML browsers, this information is not available, so it has to be estimated.

An assumption was made that the projection matrix is of the following format:

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ c & d & e & 1 \\ 0 & 0 & 0 & f \end{pmatrix}$$

After transformation by the matrix and perspective division, a 3D point $(x, y, z, 1)$ is transformed into (s, t, z') like this: $\left(\frac{a}{f}\frac{x}{z}, \frac{b}{f}\frac{y}{z}, \frac{c}{f}\frac{x}{z} + \frac{d}{f}\frac{y}{z} + \frac{e+1}{f}\right)$. s and t are pixel coordinates in the texture plane. z', Z buffer depth, is never required, as re-projection relies on knowing the depth from other sources. Thus, only the following formulae are needed:

$$\begin{aligned} x &= zs\frac{f}{a} \\ y &= zt\frac{f}{b} \\ s &= \frac{x}{z}\frac{a}{f} \\ t &= \frac{y}{z}\frac{b}{f} \end{aligned}$$

These contain only two parameters dependant on the projection matrix, namely $\frac{a}{f}$ and $\frac{b}{f}$. These are referred to as `projectionX` and `projectionY` in the program.

These parameters are computed from a special calibration scene. This consists of a two-metre cube at a known position. Its image is captured and pixel coordinates of its vertices are retrieved. From them and from the known position of the vertices, the projection parameters are computed.

Projection parameters are stored in a file so that they need not be computed each time the application is run. They depend on the browser used, the size of the browser window and the field of view. Of these, the first two remain the same for an entire run of the application, while each impostor can potentially be rendered with a different field of view. Thus, there is a file for each browser–window size pair, which contains records for all fields of view already needed. It is a simple text file, with lines of the following format: “*fov=projectionX,projectionY*”.

If the application needs projection parameters and they are not available in the file, it instructs the applet to display the appropriate calibration scene and computes the projection. The original task is then resumed.

4.4.3 Impostor generation

The algorithm for generating an impostor is outlined in section 3.2. It is repeated here as algorithm 2, along with implementation notes.

Algorithm 2 Impostor generation

```
generateImpostor(impId) {
  data=retrieveDatabaseData();
  impostor=new Impostor(impId,data);
  applet.setPosition(data.generationPoint);
  if projectionDataAvailable(data.fov) {
    projection=getProjectionData(data.fov);
  } else {
    projection=applet.calibrate(data.fov);
  }
  applet.showBorderGuidanceLine(data.impostedBorder);
  findTextureArea(screenCapture());
  applet.hideBorderGuidanceLine(data.impostedBorder);
  for each brd in impostor.borders {
    applet.blackinBorder(brd);
    findBorderContours(brd,screenCapture());
    applet.blackoutBorder(brd);
  }
  impostor.assignPixelsToBorders();
  applet.blackinSectors(impostor.impostedSectors);
  createTexture(screenCapture());
  applet.blackoutSectors(impostor.impostedSectors);
  impostor.computeGeometry();
  upload(impostor);
}
```

At the start of the algorithm, the entire scene is blacked out (hidden). As the application needs to capture parts of the scene, it instructs the applet to black them in (show them) and after they are captured, to black them out again.

Whenever actual geometry is displayed for capture, a way is needed of separating it from the background. As nothing is known about the models, using a pre-defined colour for background could lead to problems if the colour was also used in a model. Thus, it was decided to capture each required view twice, changing the background colour between the snaps. This allows the geometry to be extracted from the background easily.

Note that all computations in the impostor generation process are carried out in a coordinate system with axes parallel to those of the coordinate system of the camera.

Texture area Texture area is computed by displaying a guidance line on the imposted border. A guidance line is a 1-pixel line between the border's vertices exactly as they are specified in the database. Using this line, the position of the imposted border in the texture is computed.

The texture's bottom edge is placed so that the texture doesn't extend below the lower point of the guidance line. The left-hand and right-hand edges are placed either on the guidance line's corresponding endpoints, or on the edges of the VRML window, as per the value of `side_cutoff`.

Any future screen captures only capture the texture area, to minimize handled image size.

Border flanks As described in section 3.2.2, there are serious problems with geometry extruding out of the border plane. These were solved by truncating the border at the projection of its endpoints. However, the designer can specify an alternate solution if the extruding geometry is important (it can be an entire tower, for instance).

The alternate solution lies in specifying a "flank vector" for one or both endpoints of the border. When the border is viewed from the side of the endpoint, the corresponding flank vector is used. A fake border (i. e. one not present in the database) is added to the list of borders, placed on the flank vector. The contour of the original border is then truncated to the endpoint, but the rest of it is not discarded, but used as the contour of the fake border instead. See figure 18; the original border and contour is shown in red, the flank in blue.

A zero flank border can be specified to simply prevent truncation without introducing a flank border. This is useful for situations when a piece of geometry extrudes beyond the border's endpoints but does lie in the border's plane.

Assigning pixels In order to reproject texture pixels correctly, border depth ordering has to be determined. For each column of pixels in the texture (that is, for each value of the s coordinate), the application maintains an array of all borders whose projected image includes the coordinate, sorted by distance from the camera. Note that for each value of s, the ordering can be different. Calculating the distance involves computing reprojections, so comparison results are memoized to speed up the sorting process. A description of the comparison for coordinate s_0 follows.

First, points on the borders' bottoms with coordinate s_0 are reprojected and their Z coordinates compared. If not equal, the border with higher Z coordinate is nearer.

Equal Z means the borders' planes intersect in s_0. This can only happen on border endpoints. The border which ends in s_0 is then traversed towards its other endpoint (along the s axis) and Z is compared repeatedly. This ensures consistency between adjacent pixels, the absence of which would confuse the occlusion algorithm and could result in misplaced connector faces ruining the impostor. If no difference is found after 3 steps, the borders are assumed to lie in the same plane.

Once border depth ordering is determined, computing their relative occlusion is easy. Borders are assumed to contain no holes — they occlude everything between the top and bottom contour. It is also assumed that ground is never transparent. These assumptions together imply that for each s, the list of borders at least partially visible resembles a card layout (see figure 19).

Fully occluded borders are then removed from further processing. For the rest, points of interest (POIs) are identified. These are placed on the contour, exactly as presented in section 3.2.3. For the front-most borders, POIs are also placed on their bottom edge. These will be used for floor vertices.

Texture creation All imposted sectors are blacked in and the image captured to be used as impostor texture. Any blank space on the top of the texture is clipped off. Then, the texture is reduced from the original quality to 256 colours.

The task of palette reduction has been implemented so that it is possible to use a Java class as a 'plug-in' for this purpose. Interface `ColorReducer` has been defined and any class implementing it can be specified as a parameter to the application. It will then be used in the generation process.

This was done to enable a better implementation to be used without need to modify the program's code.

The default implementation is a class named `ScaledPalette`. It reduces the number of colours by iteratively reducing the scale of values available, as described in algorithm 3.

Algorithm 3 Scaled palette

```
histogram=new Histogram;
scale=1;
for each pixel in image {
  pixel.rgb/=scale;
  if pixel.rgb not in histogram {
    histogram.add(pixel);
    if histogram.size>8*256 { //optimization 1
      scale*=2;
      for each color in histogram {
        color.rgb/=2;
      }
    }
  }
}
colorsLeft=histogram.size;
newHistogram=new Histogram;
while histogram.size>256 {
  for each color in histogram {
    colorsLeft-=1;
    newHistogram.add(color.rgb/2);
    if newHistogram.size+colorsLeft==256 { //optimization 2
      goto doneScale;
    }
  }
  scale*=2;
  histogram=newHistogram;
  colorsLeft=histogram.size;
  newHistogram=new Histogram;
}
doneScale:
for each color in histogram not yet processed {
  palette.add(color.rgb*scale);
}
for each color in newHistogram {
  palette.add(color.rgb*2*scale);
}
return palette;
```

The basic idea of the algorithm is to first construct a histogram of the image[25]. Then, all colour values are divided by 2 (rounding down). This is repeated until the number of distinct colours falls below 256. Then, colours are scaled back up and the image resampled with the new palette.

There are two optimizations introduced in the algorithm. Each scaling (division by 2) can combine at most 8 colours into a single one[26]. Thus, once more than 8*256 distinct colours are found during histogram construction, the histogram created so far can be scaled immediately, as it

[25] A hashtable is used for the histogram to ensure fast access.

[26] The colours (r, g, b), $(r+1, g, b)$, $(r, g+1, b)$, $\ldots$ $(r+1, g+1, b+1)$ merge into one, if present (for even values of r, g, b).

is certain the number of colours will not fall below 256. This is done to limit memory space required by the algorithm. Applying the scale sooner could result in obtaining fewer than 256 colours in the end (if no new distinct colours are encountered, for instance), which would mean some colours were merged needlessly. Naturally, after the histogram is thus scaled, all further pixels processed must be scaled the same before being inserted into the histogram.

The second optimization alters the final steps of the algorithm. Once the histogram is complete and is being scaled iteratively, the scaling is carried out one colour after another. Once the number of distinct colours falls so that when added to the colours still unscaled they total 256, the scaling is ended and the remaining colours are left in one-step-better quality. It does make the algorithm dependent on the order of colours in the image, though.

Connector faces Vertices for the impostor are placed according to POIs. The mesh is then refined so that vertices are arranged in quads, making border face creation trivial.

Finding where to place connector faces is a more difficult task. A connector face must be anchored in vertices already present in the mesh. The texture is processed in columns. A connector face should be present at coordinate s_0 where a border b loses visibility (it becomes occluded or ends altogether). First, all borders in $s_0 + 1$ are considered as partners for b for anchoring the connector face. Figure 20 shows all possible configurations of POIs in b and a candidate border b_c. Cases (b) and (c) are not usable for anchoring a connector face. In case (a), which is the most common one, the connector face is anchored in b and b_c. If no candidate border is applicable, the border b_t, immediately occluded by b, is chosen (if it exists).

If an anchor b, b_a is found, the connector face is created. It is a quad with vertices in b's top and bottom contour points in s_0 and b_a's top and bottom contour points in $s_0 + 1$.

Horizontal connector faces are also added in situations like those in figure 21. A border keeps a list of all borders which it occludes partially and the horizontal connector face simply connects all relevant POIs of these borders to the occluding border.

Floor Vertices forming the floor are already computed, they just have to be triangulated. As with colour reduction, the triangulation has been delegated to an external class implementing the interface `Triangulator`. Thus, a better algorithm than the default one can easily be supplied.

The default implementation is class `DaCDelaunay`, which performs a divide-and-conquer implementation of Delaunay triangulation. The Java code for this was obtained from [4]. The `Triangulator` interface allows for specifying which edges should be part of the triangulation (edges representing the borders), but the default implementation disregards this constraint. Using code for constrained Delaunay triangulation was also considered, but all available samples were licensed under GPL[27] and it was decided to avoid the necessity of placing the entire program under GPL.

4.4.4 Impostor placement

Impostor placement is a process which takes a border ID as input and determines the generation point and camera orientation for generating an impostor for this border. The camera orientation should be close to the direction from which the impostor is most likely to be viewed. The generation point's position relative to the border affects texture quality of the resulting impostor.

The entire aim of the placement process is to find the best-looking impostor for the border. Thus, it is a difficult task for automation. An attempt was made to establish some optimization

[27] GNU General Public License; available from `http://www.gnu.org/licenses/licenses.html#GPL`

Figure 18: Splitting a flank

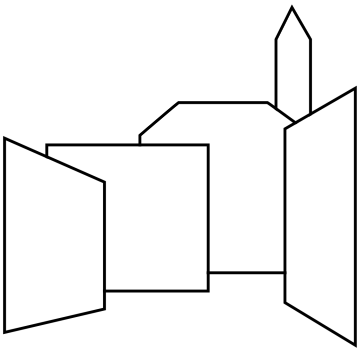
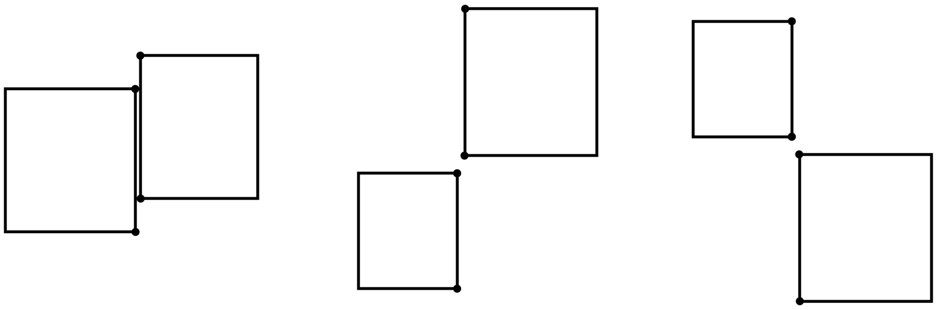


Figure 19: Border occlusion



(a) (b) (c)

Figure 20: Connector face anchoring configurations

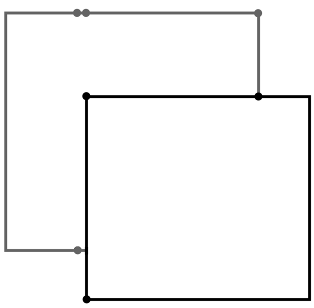


Figure 21: Horizontal connector face

criteria and base an algorithm on them. However, it is not fail-proof. Thus, for some complicated situations, better results are obtained placing the impostor by hand. Most of the time, though, the automated process produces sufficient results.

The second important part of the placement process is determining which sectors the impostor imposts and which borders are not visible. This can be carried out automatically even for impostors placed by hand.

The first part, finding the generation point, is outlined as algorithm 4, with implementation notes detailed below. The algorithm differs in several places according to whether a normal impostor is being placed or a surround-type one (which is denoted by the fact that the imposted border is marked as inverted).

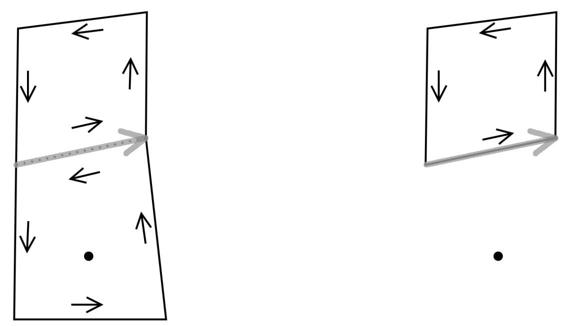
Algorithm 4 Impostor placement, part I: Finding the generation point

```
// camera orientation
computeCameraOrientation();
// side cutoff
for each wing in mainBorder.wings {
  if wing.isBlocking and angle(mainBorder,wing)<90+tolerance {
    side_cutoff.add(sideOf(wing));
  }
}
// generation point position
// y
generationPoint.y=max(mainSector.vertices.y)+avatarHeight;
// z
generationPoint.xz=point in distance dist from mainBorder;
applet.blackinSector(mainSector);
applet.setPosition(generationPoint)
while geometryTopsAreClipped(screenCapture()) {
  generationPoint.z+=step;
  applet.setPosition(generationPoint);
}
// x
minX=limit on S coordinate of mainBorder.b;
maxX=limit on S coordinate of mainBorder.a;
if minX>maxX {
  adjust(generationPoint.x,minX,maxX);
}
for each x in [minX,maxX] {
  fitness[x]=0;
  for each brd in mainSector.borders {
    fitness[x]+=crit(brd,x);
  }
}
generationPoint.x=argmax(fitness[x]);
```

Definitions Several terms have to be defined to make the algorithm description easier.

Imposted border The border specified as parameter to the placement process — the one for which an impostor is being calculated.

Main border For a surround-type impostor, it's the imposted border. For a normal impostor, it is the other orientation of this border (see figure 22). In other words, the main border is the version of the imposted border classed as inverted (i. e. seen from outside).



Normal impostor Surround-type impostor
(generation point marked as a dot)

Figure 22: Main border

Main sector The sector to which the main border belongs.

Generation border This is only defined for normal impostors, as identical to the imposted border. Basically, it is the border opposite to the main border, which does not exist for surround-type impostors.

Generation sector The sector to which the generation border belongs.

Wing A wing of border b is a border immediately preceding or following b in its position on sector edge.

Camera orientation For a surround-type impostor, the view direction is simply perpendicular to the imposted border. For a normal impostor, camera orientation is based on the direction of borders adjacent to the imposted border.

First, wings of the generation border are considered, as they naturally correspond to directions from which the impostor is likely to be looked upon. Each wing within a threshold angle of being perpendicular to the generation border is used for the view direction (if both satisfy this condition, they are averaged). If none of them is applicable, the same test is repeated for the main border's wings. If this does not yield a direction either, a look perpendicular to the imposted border is chosen.

Figure 23 shows some typical topologies, with wings used for determining the view direction highlighted.

All further computations are carried in out in a coordinate system with axes parallel to those of the coordinate system of the camera.

Side cutoff `side_cutoff` determines which edges of the impostor can be truncated. The calculation depends on the main border's wings, as these can potentially limit visibility to the sides, which warrants a cutoff. First, the wing has to be capable of blocking view. This means it's a border of type `wall` which has not been marked as `non-blocking` in the database (borders whose geometry is too low would probably be marked as such). Second, the wing must not open up the

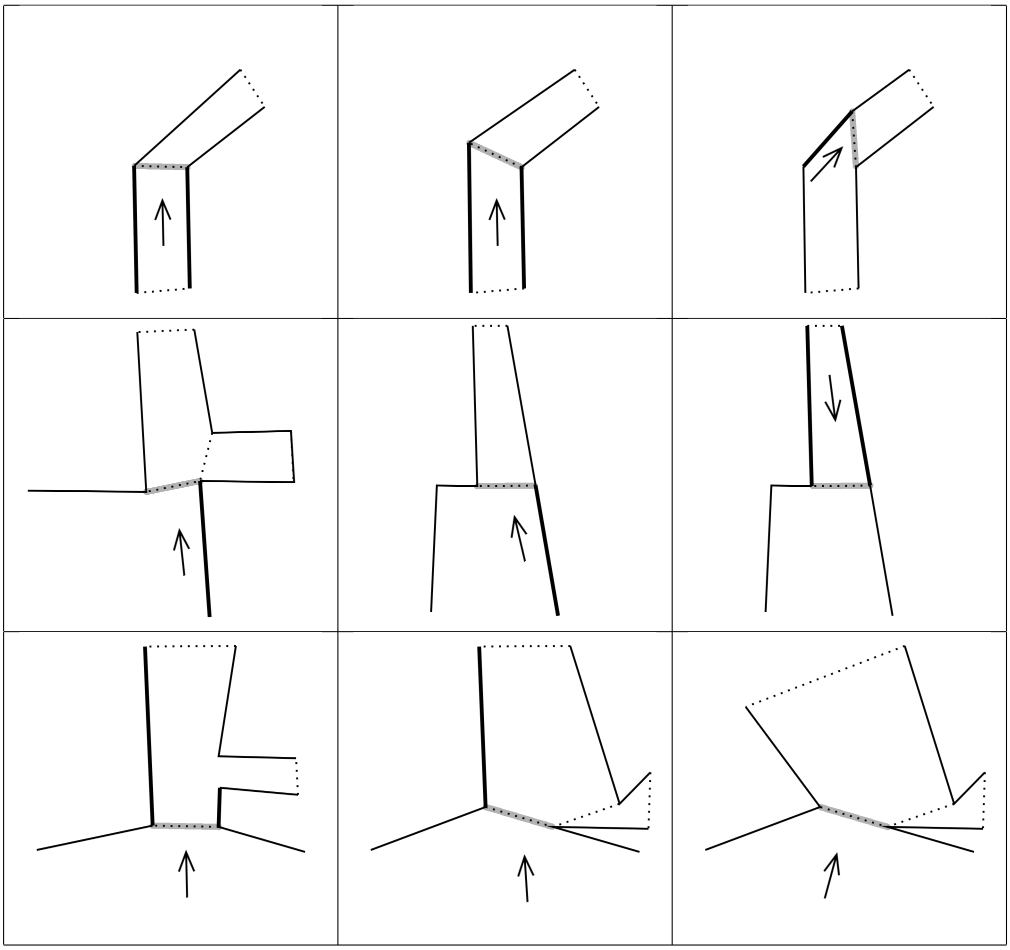


Figure 23: Determining camera orientation

view too much. This is measured by comparing the angle between the wing and camera orientation to a threshold. An illustration of `side_cutoff` is given in figure 8 on page 28.

Generation point position

Camera Y The Y coordinate of the generation point is obtained simply by placing the avatar on the highest-positioned vertex in the main sector. This was chosen on the grounds of being simple. In most cases, it also helps to prevent the impostor being viewed from above, for which the impostors are not designed.

Camera Z The Z coordinate determines the distance of the camera from the imposted border. The camera should be placed as close to it as possible, to preserve maximum detail in the impostor texture. However, it must be ensured the geometry of the main sector does not extrude above the browser window[28]. Otherwise, it would appear clipped in the impostor, which is an immediately visible visual defect.

Unfortunately, computing the Z coordinate precisely would require knowledge of the geometry data. As this data is stored in arbitrary VRML files, however, this knowledge cannot be obtained. Attempts were made to use bounding box information as stored in the database for estimating geometry heights, but it was found the values given are too inaccurate for this purpose, resulting in placing the generation point too far and thus in needlessly poor quality of the impostor texture.

A different approach was chosen, namely approximating the distance from rendering the geometry. First, the camera is positioned a set distance from the border, and image of the main sector rendered from this point is captured. If there is geometry on the top edge of the browser window, it is assumed to be clipped and the camera is moved back by a set distance. This process is repeated until all geometry fits beneath the top edge of the browser window.

In the actual program, the process is split into two parts to speed it up. When the camera is being moved back, it is moved by quite a large distance (default 10 metres). When a point is reached where the geometry fits, an attempt is made to move the camera forward again, this time in smaller increments (default 2 metres). This is repeated until the geometry becomes clipped again, resulting in approximating the ideal distance within a tolerance equal to the smaller increment.

Camera X The most important effect of the X coordinate of the generation point is the angle under which borders are viewed. This matters especially for borders nearly parallel to the viewing direction (referred to as "longitudinal borders"), which are highly susceptible to perspective distortion. It is desirable to ensure they are viewed under the maximum possible angle.

The angle under which a longitudinal border is seen translates directly into the width of its projection, which can be computed more easily than the actual angle. Thus, the task transforms into finding an X position for which the widths of visible borders are maximal. This should only consider longitudinal borders, as borders more or less perpendicular to the viewing direction preserve good quality of the texture even when viewed under a small angle. Rather than classify the main sector's borders as being longitudinal or not, the criterion has been modified to maximizing a sum of logarithms of the projection widths. The more longitudinal a border is, the shorter its projection width is and thus a one-pixel increment in width translates to a larger increment in the

[28]Theoretically, all imposted sectors should be checked, not just the main one. In practice, however, the projection of their geometry is so much shorter because of distance that it cannot be clipped.

logarithm, precisely what is needed. From some positions, some borders can be inverted (i. e. not visible at all), which should be avoided if possible. Thus, a penalty is awarded for each inverted border. This combines into the following criterion:

$$\begin{aligned} x_{cam} &= \arg \max_{x \in [x_{min}, x_{max}]} \left\{ \sum_{B \in borders} crit(B, x) \right\}, \\ & \quad \text{where} \\ crit(B, x) &= \begin{cases} \log\left(width(B \text{ projected from } x)\right) & \ldots \quad B \text{ is not inverted} \\ \\ -\log\left(width(\text{VRML window})\right) & \ldots \quad B \text{ is inverted or 1 pixel wide} \end{cases} \end{aligned}$$

No simple way of computing the optimal x_{cam} was found, so it is searched for by brute force — the criterion is calculated for each point in the range $[x_{min}, x_{max}]$ (in 0.1 metre increments). Whether a border B is inverted or not is determined by pre-computing intersections of the straight line containing the border with the X axis. This point divides the axis into two segments; in exactly one of them, B is seen as inverted.

The values of x_{min} and x_{max} are chosen to limit the projected position of the imposted border's starting and ending vertices. Foremost, the entire imposted border must fit into the VRML window. For sides not in `side_cutoff`, the vertex's projection must not fall too close to the window edge, to allow for a reasonable amount of geometry visible on that side.

If the camera is too close to the imposted border, it can happen that $x_{min} > x_{max}$. In such a case, the camera is moved further back. There are two alternatives, specifiable by a parameter of the application. The first one adjusts z_{cam} only so that $x_{min} = x_{max}$. The second one moves it slightly further, to allow for some optimization in the X coordinate (as described above).

Imposted sectors & borders Determining which sectors are imposted by the impostor is the second part of the placement process. Marking a sector as imposted has two consequences: the sector is hidden when the impostor is shown in the scene and the sector's borders are processed when the impostor is being generated. Because the placement process knows nothing about geometry, it marks many sectors as imposted (and thus potentially contributing their geometry to the impostor), even though some of them are totally occluded. The only negative effect of this is increased generation time for the impostor. Again, it is possible to improve the results by hand, this time removing sectors which will clearly not affect impostor geometry due to occlusion.

There are several flags of sectors and borders involved in the process. These are specified in the database and denote additional relevant information which the designer has specified for the sector or border. Algorithm 5 describes the process. Several implementation notes are detailed below. Note that computations are carried out in world coordinates.

Imposted sectors First, a set of half-planes whose intersection delimits the impostable area is constructed, as illustrated in figure 24. The first two half-planes correspond to edges of the field of view — their intersection defines the field of view. A third half-plane corresponds to the imposted border[29]. If an edge is in `side_cutoff`, a half-plane excluding geometry beyond the corresponding wing is added to the set. The texture area of the future impostor is also computed.

[29]The plane's edge is shifted slightly further than the imposted border, to prevent mistakenly identifying sectors like A as imposted.

Algorithm 5 Impostor placement, part II: Finding imposted sectors

```
// imposted sectors
halfplanes.add(leftFovEdge.rightHalfplane);
halfplanes.add(rightFovEdge.leftHalfplane);
texRange=VRMLwindow.width;
if "left" in side_cutoff {
  halfplanes.add(line(mainBorder.b,mainBorder.leftWing.a).rightHalfplane);
  texRange.min=project(mainBorder.b).s;
}
if "right" in side_cutoff {
  halfplanes.add(line(mainBorder.a,mainBorder.rightWing.b).leftHalfplane);
  texRange.max=project(mainBorder.a).s;
}
for each sector sec in database {
  if sec!=generationSector and not sector.interior {
    for each brd in sec.borders {
      if brd.distantVisible or distance(brd,generationPoint)<threshold {
        if brd is behind impostedBorder and
          brd intersects all halfplanes and
          project(brd) intersects texRange {
          impostedSectors.add(sec);
          next sec;
        }
      }
    }
  }
}
// inverted/hidden borders
for each sec in impostedSectors {
  for each brd in sec.borders {
    if brd intersects all halfplanes {
      if inverted(brd) {
        invertedBorders.add(brd);
        if not brd.invertable {
          hiddenBorders.add(brd);
        }
      }
    } else {
      hiddenBorders.add(brd);
    }
  }
}
```

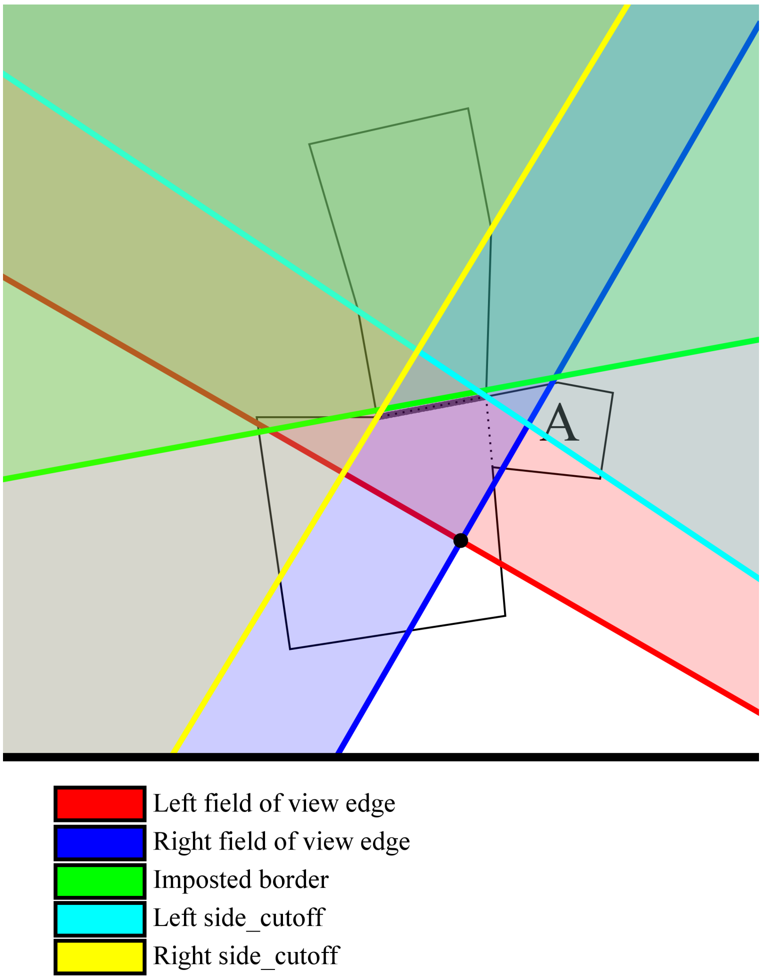


Figure 24: Half-planes delimiting impostable area

Then, all sectors are processed. A sector must fulfil all of the following conditions to be imposted:

- It is not marked as interior.
- It lies at least partially in the intersection of all the half-planes described above.
- It is not too distant from the generation point or it contains a border marked as visible from any distance.
- At least some of its geometry falls into the texture area.

Borders All borders of the imposted sectors are processed. If a border lies outside the half-plane intersection, it is marked as hidden.

A border is inverted if the generation point lies to the right-hand side of the line containing the border (looking in the direction of the border). This is computed as belonging to the appropriate half-plane. A border can be marked in the database as visible even when inverted. If that is not the case, an inverted border would be backface-culled during rendering, so it is marked as hidden as well as inverted.

4.4.5 Changes to the database

This section describes changes to the database made for the impostor generation and placement processes. All of these changes were added as new tables.

`border_flank (border_id, part, prolong, vector_x, vector_y, vector_z)`

This table specifies additional geometry information for borders which must not be truncated during impostor generation (see sections 3.2.2 and 4.4.3). `part` is either 'a' or 'b' and identifies the border's point whose extrusion is being described. `prolong` can be either 0 or 1. A value of 1 indicates the border will not be truncated, with points extending beyond the endpoint being considered part of the border's plane. When `prolong` is 0, the vector coordinates specified are used as the flanking vector. Note that the orientation of the flanking vector must correspond to border orientation. That is, to be visible, the vector's projection must point from right to left.

`place_impostor_borderflags (border_id, flags)`

Border flags used during impostor placement (see section 4.4.4) are specified in this table. The following flags can be specified:

`non-blocking` If the border is a wing of the main border, its side is not placed in `side_cutoff`.

`invertable` The border is not marked as hidden even if it is inverted.

`distant-visible` The border's sector is included into imposted sectors regardless of distance from generation point.

`tall` This flag is currently unused.

5 Conclusion

5.1 Performance & test results

VOP with the addition of interiors and impostors, as well as the Java program, were subjected to several performance tests. All of these tests were carried out on the following configuration:

- AMD Athlon 3500+ processor, frequency 2.21 GHz
- 1024 MB RAM
- NVIDIA GeForce 6800 GT, 350 MHz, 256 MB DDR3 memory at 1 GHz
- Screen resolution 1024×768
- Microsoft Windows XP Professional
- Cortona VRML browser, version 4.2
- Microsoft Internet Explorer 6.0
- Sun Java 1.5.0_06 (running the stand-alone application)

Impostor placement A total of 48 impostors were to be added to model, of which 3 are surround-type impostors. Generation point positions for all the impostors were first established running the automated impostor placement process. In 32 cases, the resulting generation points were usable. 27 of these produced fitting impostors of good quality. For the other 5, approaching distance for the limited visibility regime had to be increased, as the impostor quality was too low for being viewed from the default distance of 25 metres; still, these impostors are fully usable in the edge-only regime.

The automatic placement process gave unusable results in 16 cases. 2 of these were surround-type impostors. 1 was an impostor intentionally designed for a different set of angles than those computed by the placement process. Most of the other wrongly placed generation points were for impostors around Wallenstein's square, where gates between large open areas are common. 6 of these were positions where the impostor is viewable from an open area and thus from a wide range of angles, which places very high demands on impostor quality and thus generation point precision.

3 other mis-placed impostors were on borders where the view opens up, which is again a more difficult situation to assess as the right amount of geometry in the opening area has to be included in the impostor. In one of these cases, the placement process determined the viewing direction correctly and only the generation point's coordinates were wrong.

All but one of the impostors whose approaching distance had to be increased also involved a view to or from an open area. In total, there were 27 impostors with a large open area around their border. Of these, 11 were misplaced. Thus, it seems safe to conclude that the automatic placement process performs well in a tight environment, as only 5 impostors in such an environment were misplaced. Open areas present more problems, as there is little to base the viewing direction on, and determining the viewing direction wrongly most likely renders the impostor unusable. Even so, the process was successful in more than half such cases. Table 1 summarizes the performance of the automatic placement process. It is also worth noting that for one of the 16 misplaced impostors, a suitable generation point was not found even by hand, so the border does not include

	Tight	Open	Surround	Total
Total	21	24	3	48
Automated OK	15	11	1	27
Automated w/distance adjusted	1	4	n/a	5
Manual OK	5	7	2	14
Manual w/distance adjusted	0	1	n/a	1
Unusable	0	1	0	1

Table 1: Impostor placement results

an impostor in the end. For one of the hand-corrected impostors, approachable distance then had to be increased.

As all the impostors involved were computed for the model of a real city, the results presented are relevant for practical use. The automatic placement process is usable, having placed generation points successfully in more than two third of the cases where impostors were actually used. Still, its outputs should be verified by a human designer, especially if surround-type impostors or open areas are involved.

Impostor generation Of the 48 borders identified for impostor creation, 47 were found actually suitable for displaying an impostor. All of these impostors were generated by the `ImpGen` program. Imposted sectors were determined by the second part of the automated placement process, even for those placed by hand. The results were only edited by hand for two impostors, both of which involved a nonstandard circumstance[30].

Under these conditions, generating all 47 impostors with a 750 ms VRML delay timeout took a total time of 57 minutes on the above detailed machine, averaging at about 73 seconds per impostor. This time can be reduced by manually reviewing the list of imposted sectors for each impostor and removing those which are totally occluded and thus do not contribute to the impostor geometry. This was also done and the impostors generated again, this time in 31 minutes, which averages to 40 seconds per impostor.

Considering the impostor generation is a one-time process carried out at design time, its speed was deemed acceptable. Most of the time is spent in safety timeouts waiting for VRML actions to complete, so the process is not as computation-heavy as the time it takes would suggest.

Table 2 lists various data on impostor size for the 47 impostors generated. A typical impostor can easily be replacing 10 or more `lnodes`, while those which represent views down a long street or similar can be imposting 20 or more. As most house textures in VOP have over 10 kB, impostors clearly bring a saving of data transferred over the network and stored in memory. Triangle counts for VOP models are difficult to obtain, but an ornate façade crest or indented windows can bring the number well over 20. It is also important to remember that the triangle count given for impostors in table 2 includes floor. Thus, it seems safe to conclude that impostors reduce the number of polygons rendered as well.

Performance tests Several performance tests were carried out on various scene configurations. The results are given in table 3. The tests were taken on the above specified machine running all relevant servers (HTTP server, MySQL server, VOP system) locally.

[30] One included a high tower, resulting in the generation point being placed extremely far, the other was a surround-type impostor.

	Mean	Median
Texture size (kB)	35.6	31.6
File size (kB)	21.0	19.6
Total size (kB)	56.7	56.6
Triangle count	412.6	375

Table 2: Impostor sizes

Framerate displayed by Cortona was used as the data. The tests were taken by walking two fixed routes at constant walking speed. The first route led from Letenská Street to the end of Nerudova; it was chosen because it includes many impostors and is more or less straight, allowing for walking constantly without the need to stop and turn around. The second route concentrated on turning and involved doing a complete circular walk around the block separating the upper and lower part of Malostranské náměstí.

All configurations used share these settings: do not use LOD; use highest detail representation (L1); VRML only page layout. The configurations were the following:

25m Limited visibility regime with standard approaching distance to impostors (25 metres).

Appr Limited visibility regime allowing approaching the very edge of the impostor. This mode is not intended for normal browsing, as it exposes the poor close-up visual quality of impostors. However, it was included in the tests for comparison with the standard limited visibility regime, as it does not involve proximity sensors on impostor gateways.

Edge0 Edge-only regime, with visibility set to 0, only showing immediately adjacent sectors.

Edge20 The default setting, edge-only regime with visibility set to 20.

Edge50 Edge-only regime, visibility set to 50.

Vis20 Impostors disabled, visibility 20.

Vis50 Impostors disabled, visibility 50.

Vis254 Impostors disabled, visibility 254.

All The entire model loaded.

Visibility 20 was chosen as being the default value in original VOP. Visibility of 50 was included because in situations where a view into the distance is not possible (i. e. occluded), visibility 50 and disabled impostors seem to offer visual results comparable to impostor use. In places where a view into the distance (such as down a long street) is possible, impostors naturally offer a more realistic experience. Visibility of 254 was included as the highest possible visibility value short of loading the entire model.

All of the configurations were tested twice, once with the DirectX renderer (labelled 'HW') and once with Cortona's software renderer (labelled 'SW').

Several points can be inferred from the figures.

- When using visibility 0, difference between the two impostor regimes is minimal. Nevertheless, as the differences between *Appr* and *25m* show, the necessity of employing proximity sensors in the limited visibility regime does have an impact on performance.

Mean FPS	HW Straight	HW Round	HW **Total**	SW Straight	SW Round	SW **Total**
25m	72.5	78.7	**76.3**	31.8	33.8	**32.8**
Appr	74.0	85.3	**79.6**	33.2	36.4	**34.8**
Edge0	79.1	81.3	**80.2**	30.9	34.4	**32.5**
Edge20	65.5	78.5	**72.6**	28.4	30.0	**29.4**
Vis20	74.4	76.5	**75.8**	28.9	31.8	**30.0**
Edge50	50.7	46.9	**48.9**	23.1	22.0	**22.5**
Vis50	73.6	45.0	**59.0**	23.5	22.9	**23.2**
Vis254	41.5	32.8	**37.6**	19.2	17.8	**18.6**
All	8.0	8.8	**8.3**	7.2	7.8	**7.4**

Table 3: Performance test results

- The addition of impostors places increased requirements on performance, which can be seen as *Edge20* and *Edge50* offering worse performance than *Vis20* and *Vis50* respectively. However, these comparisons completely disregard the much better visual quality brought by the use of impostors.
- As far as visual quality is concerned, *Edge20* can be deemed about equivalent to *Vis50* in close quarters. Where views into the distance are possible, an impostor for that view can easily surpass even the visual quality offered by *Vis254*. From this, it is obvious that impostor use brings considerable improvement in rendering speed.

Table 4 expresses the performance with impostors relative to selected other configurations. Setups offering comparable visual quality are marked in bold. It can be seen that the default setup, *Edge20,* offers a 25% performance gain over *Vis50* while most of the time providing better visual quality.

	HW Vis20	HW Vis50	HW Vis254	SW Vis20	SW Vis50	SW Vis254	Average Vis20	Average Vis50	Average Vis254
25m	**101%**	130%	203%	**109%**	141%	176%	**105%**	135%	190%
Edge0	**106%**	136%	213%	**108%**	140%	175%	**107%**	138%	194%
Edge20	96%	**123%**	**193%**	98%	**127%**	**158%**	97%	**125%**	**176%**
Edge50	65%	82%	**130%**	75%	97%	**121%**	70%	90%	**126%**

Table 4: Relative performance test results

5.2 Conclusion

The task of this thesis was to add support for interiors and impostors into the existing Virtual Old Prague project. Both of these goals were accomplished.

Interiors Interiors were implemented by basing their design on existing VOP structure and code, so that much of original functionality can be used on interiors without modification. Support was added for doorways which allow a user to enter and exit interiors by clicking on an appropriate entrance in the scene. Sectors were extended to include lighting and automatically generated ceilings. A new type of border was added for interior walls. Their geometry is generated by the system and they can be textured by a repeating texture, by a single large texture, or rendered in flat colour. They can include windows or doors. Doors are used to connect interior sectors, utilising the existing mechanism of gates, extending it beyond use on `proxi` borders.

Impostors Depth-augmented impostors were added to the system. Compared to a traditional flat-pane impostor, these can be viewed from a substantially greater range of angles and positions. They also offer limited parallax effects and self-occlusion.

Impostors were linked to gate borders, as these naturally correspond to places where views into the distance exist. Two regimes were introduced for displaying impostors. The limited visibility regime tries to display impostors in preference to actual objects whenever possible, resulting in the least amount of geometry being rendered at any one time. It relies on the existence of an impostor for almost every gate, as it only ever loads actual geometry for sectors immediately adjacent to the one the user is currently in. This regime offers slightly better performance but worse visual quality, making it better suited for low-performance client machines. The edge-only regime offers better visual quality at the cost of rendering more geometry, as it displays a wider area around the user in full geometry and only uses impostors for views beyond this area.

Apart from gate-linked impostors, so-called "surround-type" impostors were also added. These can be bound to any border and are normally used for views outside from an interior.

The system offers means of generating impostors from the model automatically. However, it is also possible to include hand-made impostors into the model. This allows for creation of impostors for parts of town not present in the model (most likely using a real-world photography for texture).

Impostor generation Due to the absence of a suitable programmable environment, data for impostors is obtained by capturing the output of a VRML browser from screen. A program was developed which bypasses the need to access the renderer's internal data structures by displaying the scene per borders, processing the image data obtained to infer impostor geometry. The impostor generation process is based on a method described in [5]. Instead of employing a regular grid of augmenting vertices, it utilises its knowledge of geometry inferred from the image data to refine the vertex mesh locally as necessary. It includes an algorithm with implementation by Lambert [4].

The program also offers a simple authoring capability, automatically determining the generation points from which impostor views should be captured. Choice of such points requires understanding of the city's topology as well as intuition and aesthetic feeling. Thus, the automatic process is not entirely reliable. In generating the impostors added as part of this thesis, it failed in determining a usable point about 30 percent of the time. It can thus be used as a tool to help the designer, but its outputs must be checked before use.

Sample interiors Two interiors were added to the model to demonstrate interior presentation. One is a small part of the building of the faculty of mathematics and physics on Malostranské náměstí. The other is St. Nicholas church, also on Malostranské náměstí. These utilize all the interior features added. Their detailed description is given in appendix A.

Sample impostors Impostors were added to selected borders in area of Malostranské náměstí, Mostecká, Nerudova, Sněmovní, Thunovská, Tomášská, Valdštejnská and Valdštejnské náměstí. All of these impostors were generated using the program created. They include three surround-type impostors for looking out of the sample interior of MFF.

5.3 Future work

As outlined in section 1.4, the focus of this thesis was on implementation in the system, not on authoring tools. Creating such tools would be the logical next step. Interior design could be integrated into the existing VOPedit program. A tool for visual presentation and possible editing of impostor placement could also be created.

The generation program was designed with two plugin points: triangulation and colour reduction. Fully usable implementations were provided for both of these, but a more sophisticated colour reducer could be considered.

The automated placement system was designed primarily as a simple helping tool. Its performance could probably be improved by performing a detailed analysis of the ideal generation point positions as determined by a designer and inferring more accurate rules from it. It might also be subject to further research if some form of pre-determining occlusion could be implemented in the present environment of minimal knowledge of actual geometry. This could potentially reduce the number of sectors identified as being imposted.

A Added building interiors

A.1 Faculty of mathematics and physics

A small part of the building of MFF was modelled to demonstrate the various interior-related features. The entrance wall is a hand-modelled `lnode`, all other walls are `inwalls`, generated by the system. Walls in the entrance hall are rendered in flat colour, the rest of walls is textured with a repeating wallpaper.

The arched ceiling in the entrance hall is implemented as a stand-alone `node`, the rest of the ceilings is generated automatically. The ceiling of "Rotunda" (the room with columns) contains a roof window, to demonstrate the possibility of specifying a custom shape for the generated ceiling.

All of the interior sectors have lights specified. There are also lamps with point lights added as `nodes` to the passageway. The stairs, pillars and the door frame are added using the standard `node` mechanism. Three surround-type impostors were generated for the interior.

A.2 St. Nicholas church

The church of St. Nicholas was chosen as a representation of an interior attractive for tourists. Its walls are `inwalls` with non-repeating wallpapers. The ceiling is implemented as a stand-alone `node`. The textures were acquired using a digital camera.

References

[1] Jiří Čížek, Kamil Ghais, Stanislav Mikeš, Jakub Rajnoch, Michal Holub, and Pavel Chromý. *Virtual Old Prague Documentation.*

[2] International Organization for Standardization. *ISO/IEC 14772-1:1998: Information technology — Computer graphics and image processing — The Virtual Reality Modeling Language — Part 1: Functional specification and UTF-8 encoding.* International Organization for Standardization, Geneva, Switzerland, 1998. Available from: `http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/`.

[3] International Organization for Standardization. *ISO/IEC 14772-2:2001: Information technology — Computer graphics and image processing — The Virtual Reality Modeling Language — Part 2: External authoring interface.* International Organization for Standardization, Geneva, Switzerland, 2001. Available from: `http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/`.

[4] Timothy Lambert. Delaunay triangulation algorithms [online]. Available from: `http://www.cse.unsw.edu.au/~lambert/java/3d/delaunay.html`.

[5] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):C207–C218, 1997.

[6] Maryann Simmons and Carlo H. Sequin. Portal tapestries. Available from: `http://citeseer.ist.psu.edu/466318.html`.

[7] Ann Wollrath and Jim Waldo. RMI. In *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley Professional, 1998. Available from: `http://java.sun.com/docs/books/tutorial/rmi/`.